



Security Audit Report

NANO Foundation

07-01-2019

www.red4sec.com

Contents

1. Introduction	3
2. Executive Summary	4
3. Scope and Purpose	6
4. Recommendations	8
5. Cryptography Assessment.....	9
5.1 Summary.....	9
5.2 Logical Deliverables.....	9
5.3 Coverage	10
5.4 Evaluating Nano’s Security Goals	11
5.5 Attack Scenario Evaluation.....	12
5.6 Real World Viability and Scalability Claims.....	13
5.7 Implementation Review	14
5.8 Impact and Attack Cost Evaluation.....	14
5.9 Conclusions	15
6. Network Performance Analysis.....	16
6.1 Summary	16
6.2 Nano Protocol.....	17
6.3 Performed Simulation Attacks.....	18
6.4 Conclusions	24
7. Source Code Vulnerabilities	25
7.1 Vulnerability Severity	25
7.2 Methodology.....	26
7.3 Coverage	27
7.4 Automatic Analysis.....	29
7.5 Manual Analysis	30
7.6 List of vulnerabilities	32
7.7 Vulnerability details.....	33
1 - Improper Validation of Array Index.....	34
2 - Incorrect Type Conversion or Cast.....	36
3 - Code Styling	38
8. Annexes	40
Annex A List of conducted tests	41

1. Introduction

Nano is a cryptocurrency that offers no fees, near-instant transactions and extremely high scalability. Unlike traditional cryptocurrencies which use blockchains, Nano uses a novel block lattice approach, in which each account has its own blockchain, and only the account owner can modify its blockchain.



Nano offers the following features:

- Each account has their own Blockchain.
- Wallets pre-cache the anti-spam PoW.
- Running a node cost next to nothing.
- Environmentally Friendly Cryptocurrency.
- Incredibly Lightweight with no fee for processing transactions.

This report has been performed by Red4Sec Cybersecurity as a **security audit** and **cryptographic assessment**, which covers Nano with a great focus on its cryptographic components, network and security protocols, source code and configuration errors.

This audit includes all the tests performed and vulnerabilities discovered in Nano by Red4Sec at the time of the audit.

This is a final and **complete audit** which includes:

- Nano Cryptographic Assessment.
- Network Performance Analysis.
- Source Code Audit.

All information collected here is strictly **CONFIDENTIAL** and may only be distributed by NANO with Red4Sec express authorization.

2. Executive Summary

As requested by **Nano Foundation** and as part of the vulnerability review and management process, the company Red4Sec Cybersecurity has been asked to perform a security audit and cryptographic assessment in order to assess the security of the source code.

This security audit has been carried out between the dates: **24/10/2018** and **30/11/2018**.

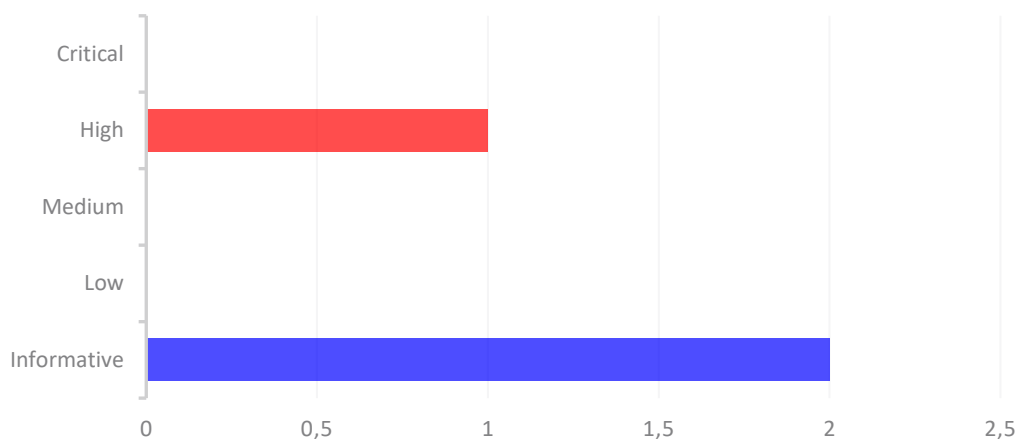
Once the analysis of the technical aspects of the environment has been completed, the performed analysis shows that the audited **source code contains non-critical vulnerabilities** that should be mitigated as soon as possible.

On the other hand, the distributed denial of service simulation reflects **great network stability** on the part of the Nano systems, although it could be affected by more elaborate attacks.

Finally, after studying the whole project it has been possible to determine that Nano presents a **proper cryptographic implementation** design.

During the analysis, a total of **3 vulnerabilities** were detected. These vulnerabilities have been classified in the following levels of risk according to the impact level defined by CVSS (Common Vulnerability Scoring System):

VULNERABILITY RESUME



Red4Sec has been able to determine that the **overall security level** of the asset is **optimal**, since no critical vulnerabilities have been detected and the existing vulnerabilities do not compromise the security of the asset and their users.

The general conclusions of the performed audit are:

- Nano presents a cryptocurrency design that does indeed achieve its goals of high efficiency, high scalability and low latency through the combination of cryptographic engineering and network engineering considerations and optimizations.
- No critical risk vulnerabilities have been detected, given that the source code of the project is correctly implemented and safe programming guides have been applied.
- Since the entire code has not been reviewed, and since total security does not exist, it cannot be guaranteed that vulnerabilities will not appear in the future.
- The structure, style and organization of the code must be improved in order to maintain the reputation and good image of the project
- None of the findings in this report constitute a serious roadblock for the real-world deployment of Nano as a cryptocurrency ledger.
- Apply all proposed recommendations considered necessary to improve the security of the Nano environment.
- In order to deal with the detected vulnerabilities, an action plan must be elaborated to guarantee its resolution, prioritizing those vulnerabilities of greater risk and trying not to exceed the maximum recommended resolution times.

3. Scope and Purpose

Nano has asked Red4Sec to perform an analysis of the project source code.

Red4Sec has evaluated the **security level** against computer attack, identifying possible design, configuration or programming errors, guaranteeing the confidentiality, integrity and availability of accessible, treated, and stored information.

The **scope** of this evaluation includes:

- **Description:** Source Code Audit – Nano.
- **Project Audited:**
 - Source Code Audit (<https://github.com/nanocurrency/nano-node>)
 - Consensus Algorithm
 - Transport Layer
 - Business Logic Vulnerability Audit
 - Check the correct function and behavior of the code.
 - GitHub Audited Commit:
`339afd767885a0f08254b3670a0c8accafb2be64`
 - Cryptographic Assessment.
 - Network Performance Analysis.

Within this Scope, Red4Sec has prioritized in the following consensus classes: ***rai::active_transactions*** and ***rai::election***.

- <https://github.com/nanocurrency/nano-node/blob/master/nano/node/node.cpp>
- <https://github.com/nanocurrency/nano-node/blob/master/nano/node/node.hpp>
- <https://github.com/nanocurrency/nano-node/tree/master/nano/secure>

The **duration** of this audit has taken around 1 month.

- **Source Code Audit:**
`24/10/2018 - 30/11/2018`
- **Final Report Documentation:**
`30/11/2018 - 05/12/2018`
- **Mitigations Review:**
`02/01/2019 - 07/01/2019`

The specific objectives of the application review have been:

1. Analyze the source code of the project, in order to detect potential vulnerabilities affecting the project, such as:
 - Input Validation
 - External calls
 - Coding best practices
 - Exception Handling
 - Control of types and default values
 - Algorithms and Cryptography
 - Logic of the Program
 - Denial of Services
 - Memory Management
 - Remote Code Executions
 - Insecure Functions
 - Manual analysis
 - Automatic analysis
 - Efficiency and Optimization
 - Code Styling
 - Non-functional requirements

In order to maintain the agreement made with the client, all those tests that could cause an interruption of any kind in the service have not been executed.

Tests have been conducted from different points of view:

- Private (Beta) Network
- Clone Server

4. Recommendations

For the resolution of the exposed vulnerabilities, the following actions are recommended:

- Solve vulnerabilities in descending order of risk and take into account the recommendations proposed by Red4Sec.
- Apply good practice techniques in source code and improve the structure, style and organization of the code as far as possible.
- Update and/or periodically patch all services, libraries, and technologies, especially those that belongs to third-parties.
- Apply safe development techniques, good practices and code styling to the entire project, before being audited and reviewed again.
- Insist on the periodic review of services, applications and source code, as well as correcting errors detected in previous reviews.

It is therefore recommended to include, in the system designs, safety requirements that can be tested in later phases to apply safe programming techniques and to introduce, in the pre-production stages, specific safety tests, such as code revisions source or the one carried out in this project.

5. Cryptography Assessment

Network security and cryptography is a subject too wide ranging to coverage about how to protect information in digital form and to provide security services. In this section, Red4Sec will be reviewing and auditing the specifications and cryptographic implementations of Nano, a feeless distributed cryptocurrency network.

5.1 Summary

Nano is a novel cryptocurrency with a “block-lattice” architecture that aims to provide higher scalability, lower latency and higher power efficiency than legacy blockchain systems such as Bitcoin.

Nano currently provides a whitepaper discussing design decisions, security goals and justifying the system against a threat model. A C++ implementation is also provided.

Red4Sec Cybersecurity has performed an audit of the Nano “feeless distribution cryptocurrency network” based on the following provided materials:

- i. Nano: A Feeless Distributed Cryptocurrency Network, Colin LeMahieu.
- ii. Nano cryptographic design elements as documented in the C++ implementation.
- iii. Special focus on certain elements of the C++ implementation.

5.2 Logical Deliverables

In this report, we aim to analyze the following objectives:

1. **Security goals:** Does Nano achieve real-world security within the security model of a blockchain-based cryptocurrency? Examples:
 - a. Do Nano wallets obtain authenticity and privacy according to the design specified in the whitepaper and provided implementation details?
 - b. Are Nano’s defenses against common attacks such as Sybil attacks adequate?
2. **Real-world viability:** Performance analysis of Nano’s real-world scalability.

- 3. Implementation design review and recommendations:** Review of the implementation best practices for Nano (cryptographic primitives, etc.) and whether these are adopted in the codebase.

5.3 Coverage

1. Nano Whitepaper

Goals:

- Higher scalability than Bitcoin.
- Lower latency than Bitcoin.
- Higher power efficiency than Bitcoin.

Components:

- Voting algorithm.
- "Block-lattice" design.
- Sequencing and confirming transactions via SYN/ACK over UDP.
- Proof of Work.
- Account management and transfers.

2. C++ Implementation

Components:

- Ed25519 with Blake2b.
- Blake2b.
- Argon2.
- Wallets:
 - Seed generation.
 - Key derivation (BIP39/44).
 - Encryption (AES-CTR).

5.4 Evaluating Nano's Security Goals

In this audit, special attention was invested in Nano's claims and security goals. While the Nano whitepaper does not explicitly state security goals, it does provide a threat model (Section V) motivated by real-world attack vector examples.

Based on these examples we assume that Nano aims to achieve the traditional security properties inherent to a blockchain-based cryptocurrency: this includes resistance to "double-spend" attacks, resistance to denial of service attacks and resistance to malicious forks.

Within that framework, we found that Nano's high-level design does achieve the nominal expected security goals of a blockchain design. We nevertheless identify the following points of discussion:

1. **"Genesis balance" sacrifices mining for efficiency.** Nano's main goals of obtaining high scalability, lower latency and higher power efficiency than other cryptocurrencies appear to be largely facilitated not simply by its innovative design but rather by the notion that a set cryptocurrency value is instantiated at the genesis of the cryptocurrency. Thus, no "mining" or generation of new cryptocurrency value ever occurs in the lifetime of the cryptocurrency.

The Nano whitepaper does not address the concern that adding mining abilities in the future might necessarily entail significant effects with regards to the design's scalability and efficiency.

2. **Voting algorithm real-world usability.** In the event of a fork, much of Nano's conflict resolution scheme relies on a voting system carried out by all nodes. There are two factors that do not seem to be fully addressed with regards the voting system proposed:
 - a. The real-world interface and usability of the voting algorithm is not studied. Will users be able to meaningfully cast informed votes every time there is a fork? How does this fit into the regular use case scenarios involving Nano?
 - b. It appears that the voting algorithm would not be effective in the common use case where a user obtains their view of the cryptocurrency "block-lattice" state through a node being provided as a web service (e.g. Etherscan for Ethereum, or any mobile/web wallet for any cryptocurrency.) In the case of a fork, a malicious third-party node is still able to present all blocks in the block-lattice

in a manner which fully satisfies the transaction verification requirements described in Section IV, Subsection I.

3. **Implicit defenses against DoS and PoW precomputation.** While Section V of the specification acknowledges multiple attack scenarios which can work in tandem to cause denial of service attacks, none of the presented defenses sufficiently rule out the threats discussed.

In particular, Section V, Subsection F, which discusses “51% attacks”, frequently makes speculative claims when discussing mitigations. Some of these concerns are explored in more detail in Section 6 of this report.

5.5 Attack Scenario Evaluation

Nano’s threat model is derived implicitly from a set of attack vectors described in Section V of the whitepaper. Having previously presented a summary of our review of the attack vectors, we now examine each attack vector individually.

1. **Block Gap Synchronization.** Given that Nano relies on UDP-based networking for high performance, the incorrect transmission of blocks may occur. In order to remedy this, the Nano whitepaper states that “a TCP connection must be formed with a bootstrapping node in order to facilitate the increased amount of traffic” required for resynchronization. Given how simple it can be for a network entity to cause UDP packet desynchronization, it is possible that the need for multiple resynchronizations could be artificially increased by an attacker as a way to exacerbate a denial of service attack. This is not currently discussed in the design and could be elaborated upon.
2. **Transaction Flooding/Penny-Spend Attack.** Spreading a single transaction against infinitely smaller microtransactions is discussed. We have nothing to add to this discussion.
3. **Sybil Attack.** While Nano’s voting mechanism does act to offset a sybil attack, we again raise the concern that no real metrics are provided with regards to the real-world usability or UX for the voting mechanism. Does it occur automatically or is it precluded by user interaction? In the latter case, how would user interaction be registered and interpreted, and how could a lack of confirmation stall further blockchain progress?
4. **Precomputed PoW Attack.** While Nano does discuss the potential for pre-calculating Proof of Work values, no real mitigation is provided.

5. **The 51% Attack.** It is not clear how Nano's voting mechanism can prevent 51% attacks unless "representative voting", discussed in Section V, Subsection F, point 3, is implemented from the beginning of the lifetime of the network. Furthermore, representative voting can indeed have the effect of rendering 51% attack-like takeovers easier, with the attacker simply focusing on the nodes with the highest levels of representative authority.

5.6 Real World Viability and Scalability Claims

Aside from the points discussed above, it was found that the Nano whitepaper adequately described a design which does achieve its real-world viability and scalability claims.

Special network-level and cryptographic engineering considerations are taken into account, and these were found to indeed help boost Nano's real-world performance benchmarks:

- A "block-lattice" design where transactions are connections between otherwise discrete blockchains.
- Communication over small UDP packets, with blockchain-level confirmations over the same medium.
- Proof of Work being restricted as an anti-spam measure instead of a "mining" mechanism.

As mentioned in both Section 5 and 6, however, it is unclear how the voting mechanism which Nano relies on would operate in real-world deployment and whether it can sufficiently scale in order to make decisions that can keep up with an arbitrary number of denial of service attacks.

This is especially important given that Nano remains susceptible to denial of service attacks as discussed in the Nano whitepaper as well as in this report.

5.7 Implementation Review

As part of the scope of this work, Red4Sec paid attention to the following cryptographic primitives as implemented in Nano:

1. **Ed25519 with Blake2b.** The original Ed25519 implementation by Daniel J. Bernstein is employed but with the Blake2b hash function used instead of SHA-3.

Critically, Daniel J. Bernstein's reference implementation performs elliptic curve operations in constant time, thereby eliminating side channel concerns. Using Blake2b instead of SHA-3 poses no security concerns and is deemed completely safe.

2. **Blake2.** The original reference C implementation written by Samuel Neves is used. This is considered to be benchmark for correct Blake2 implementations and thus no issues are found.
3. **Argon2.** Argon2 is deemed an excellent choice for this use case for the same reasons as those described in the Nano whitepaper, and the implementation is deemed safe. Again, the reference implementation is used, offsetting security concerns.
4. **Wallet key derivation.** Wallet keys are derived deterministically from 256-bit secure pseudorandom seeds. BIP39/44 is used for mnemonic seeds.
5. **Wallet encryption.** A review of the wallet encryption implementation found that while AES-CTR was used, no specific precautions seem to be undertaken for the prevention of nonce reuse.

Since AES-CTR is a stream cipher, nonce reuse under the same key can have catastrophic consequences and should be avoided.

5.8 Impact and Attack Cost Evaluation

Nano claims the following when it comes to attack costs:

"One of the advantages in RaiBlocks for using balance weighted voting is for its high attack cost; this cost is similar in Proof of Stake systems. The cost of attacking a proof-of-work protocol is in proportion to global investment in mining hardware. Given today's environment if we estimate this at \$1 billion the attack would entail making a matching purchase of hardware putting the price tag at 1\$ billion.

Balance-weighted-voting attack cost is in proportion to the total market cap. If we estimate this at \$100 billion the attack would entail buying up 50% of the market cap putting the price tag at \$50 billion. To put this another way: only if global investment in mining hardware exceeded the entire market cap of the currency itself would this attack cost difficulty flip the other direction."

While the above statement may be true in theory and especially if only Sybil attacks or 51% attacks are taken into consideration, it does not account for the potential for coordinated denial of service attacks to cause pressure on the Nano ledger and force it to capitulate to an attacker. This is especially important given the number of existing denial of service attack vectors, documented earlier in this report as well as in the Nano whitepaper.

A stronger focus on Proof of Work as well as a potential switch from UDP to TCP may act to restrict denial of service attack vectors, however the cost on the general performance benchmarks of Nano remains unclear and should be slated for future study.

5.9 Conclusions

We conclude that the whitepaper correctly presents a cryptocurrency design that does indeed achieve its goals of high efficiency, high scalability and low latency through the combination of cryptographic engineering and network engineering considerations and optimizations.

A review of the cryptographic elements of Nano's C++ implementation yielded no significant findings. Nano exclusively employs the reference implementations of state-of-the-art cryptographic primitives. Minor changes are made to the signing primitive but are deemed completely safe.

A lack of documentation was spotted with regards to the wallet encryption mechanism and this was documented in our report.

None of the findings in this report constitute a serious roadblock for the real-world deployment of Nano as a cryptocurrency ledger. That said, this report still documents underspecified shortcomings inherent to Nano, including the inability to mine additional currency and a lack of clarity with regards to the real-world functionality of the voting mechanism, which is crucial for preventing Sybil attacks or other attacks on the ledger's integrity.

6. Network Performance Analysis

6.1 Summary

Nano has a great number of mechanisms to avoid and protect from system attacks. However, the objective of the security audits carried out by Red4Sec is to analyze all possible vulnerabilities both in the logic, protocol, and to improve the level of security.

For this reason, some minor Denial of Service tests have been performed against **rai_node** service to analyze the behavior and the availability of the service when exposed to high traffic loads.

Distributed Denial of Service attacks are increasing in intensity and becoming more damaging. The disrupt of services may cause loss of revenue and reputation.

Availability is a critical aspect in any highly-distributed service, especially in blockchain-based technologies. While blockchain technologies are resistant to Distributed Denial of Services and other kind of abuses due to its distributed nature, these technologies still have weak spots that can be exploited.

This section offers a small approximation of a Distributed Denial of Service attack over Nano protocol.

These tests have been carried out by the Red4Sec team in a controlled environment, more precisely against a clone server.

6.2 Nano Protocol

In order to perform all the tests included in this report, the Nano protocol has been previously analyzed and studied.

```
struct {
    // NANO Protocol
    uint8_t magicProtocol = 0x52;
    // (0x41 Test network; 0x42 Beta network; 0x43 Main network)
    uint8_t magicNetwork = 0x41/0x42/0x43;
    // versionMax and versionMin, range of acceptable versions to relay or broadcast this
    // message to
    uint8_t versionMax;
    // version field indicates what version of the Nano protocol
    uint8_t version;
    // versionMin
    uint8_t versionMin;
    // messageType
    uint8_t messageType;
    // Extensions
    uint16_t extensions;
};
```

messageType	Name	On Bootstrap	On Realtime	Version
0x00	Invalid	Yes	Yes	0+
0x01	Not_A_Type	?	?	0+
0x02	Keepalive	No	Yes	0+
0x03	Publish	No	Yes	0+
0x04	Confirm_Req	No	Yes	0+
0x05	Confirm_Ack	No	Yes	0+
0x06	Bulk_Pull	Yes	No	?
0x07	Bulk_Push	Yes	No	?
0x08	Frontier_R	Yes	No	?
0x09	Bulk_Pull_Blocks	Yes	No	11+
0x0A	Node_ID_Handshake	No	Yes	12+
0x0B	Bulk_Pull_Account	Yes	No	12+

Using the data provided by the Nano team, the existing documentation and the wireshark filters, we have been able to build a laboratory to perform denial of service attacks, and analyze the behaviour of Nano nodes when exposed to these attacks.

6.3 Performed Simulation Attacks

After thoroughly studying the protocol, Red4Sec proceeded to perform a series of **flood tests** against the service using a burst of packets with the different protocol commands. The main objective was to check the behaviour of the nodes before a minor denial of service.

In all the tests carried out, the traffic was generated based on the standard Nano protocol (packet header, commands, extensions, ...) so that the server accepted it as legitimate traffic and tried to process it.

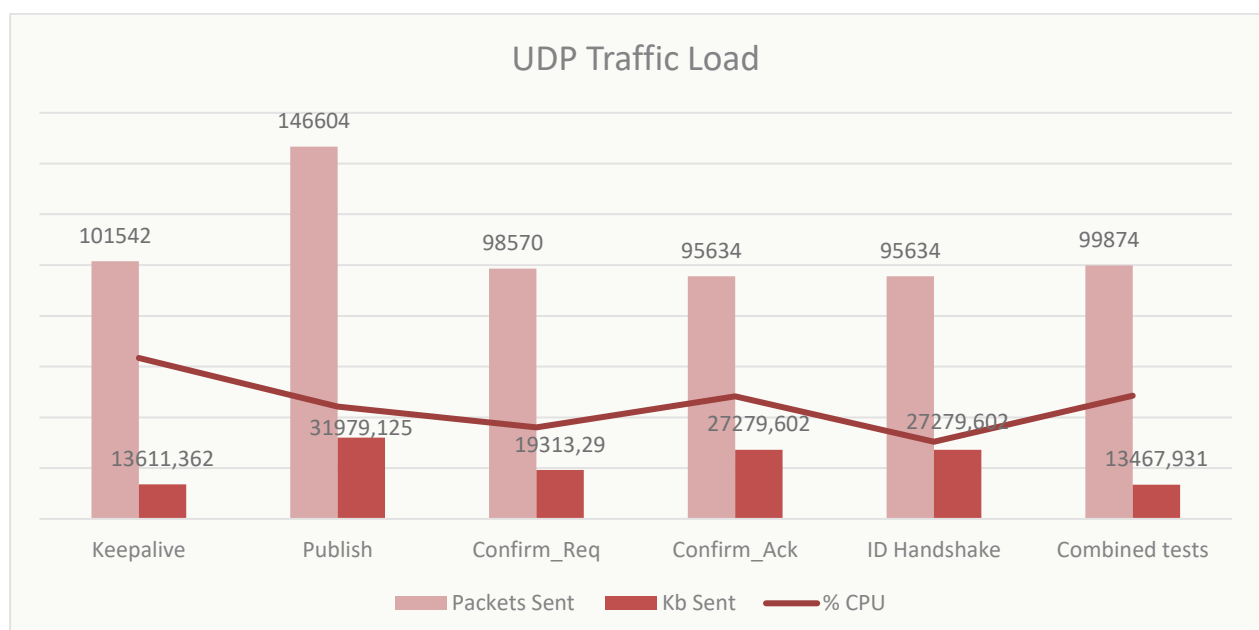
In each test, 50% of the traffic was generated conforming to the protocol specification, but with modification to some values.

The other 50% was generated under the same conditions, but instead, modified the values and their length in order to check the behaviour against incorrect values and malformed packets (packet fuzzing).

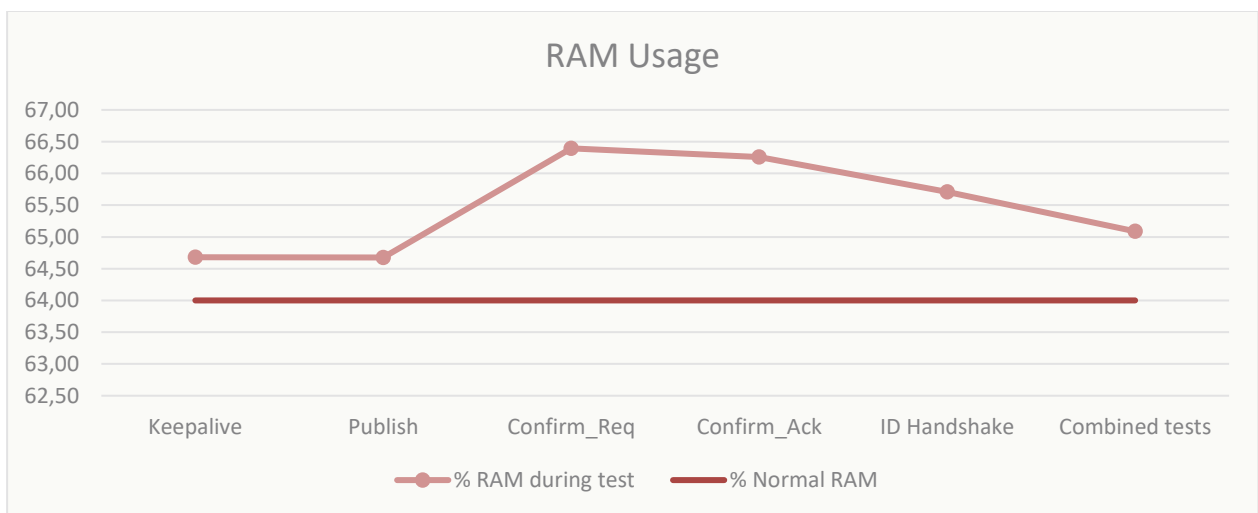
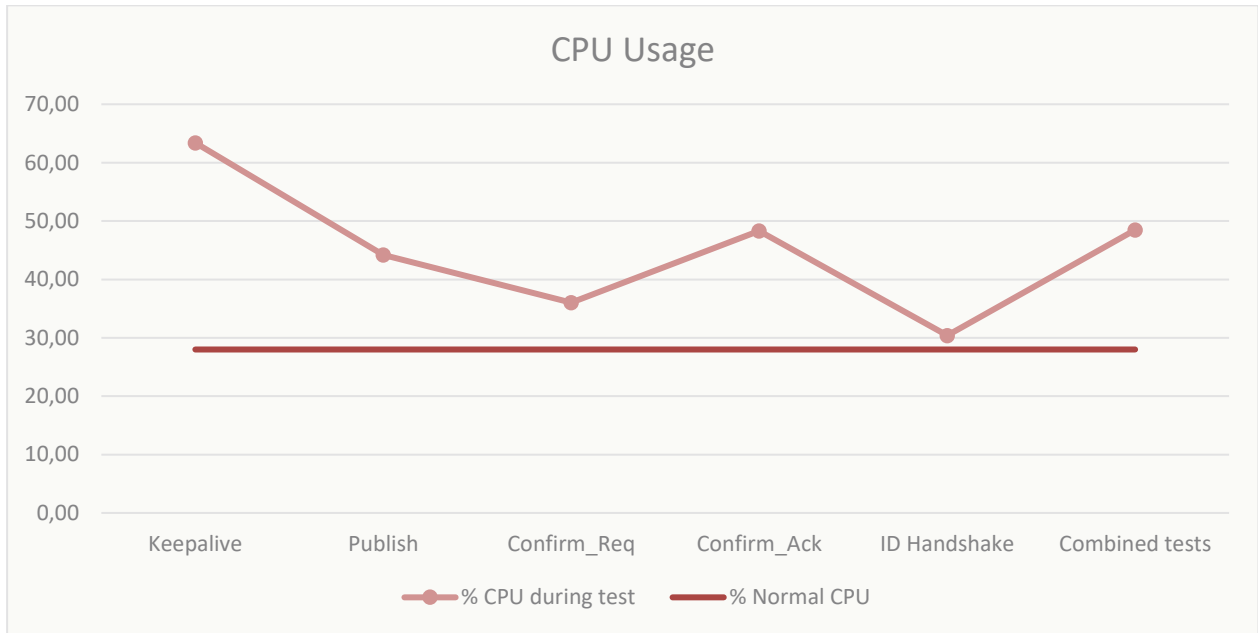
The tests have been performed over TCP (Bootstrap) and UDP (Real Time), for each command individually. On the other hand, an additional test (Combined Test) has been carried out, mixing all the commands.

During the tests, CPU and memory usage have been compared with normal values obtaining the following results.

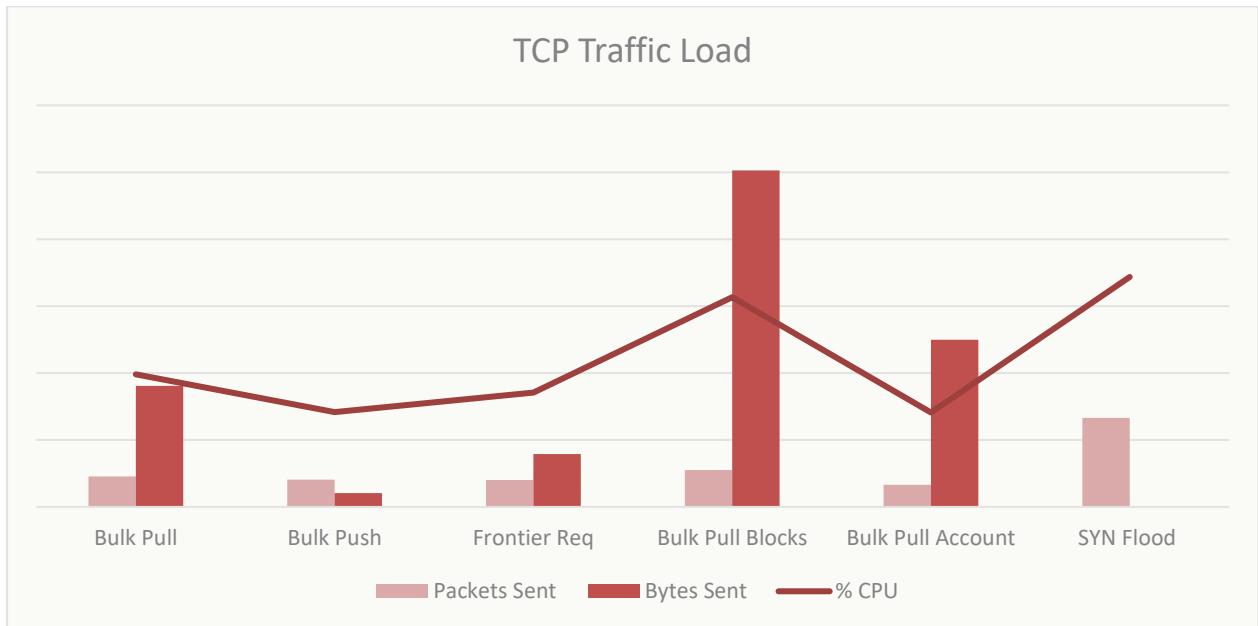
First UDP Simulation



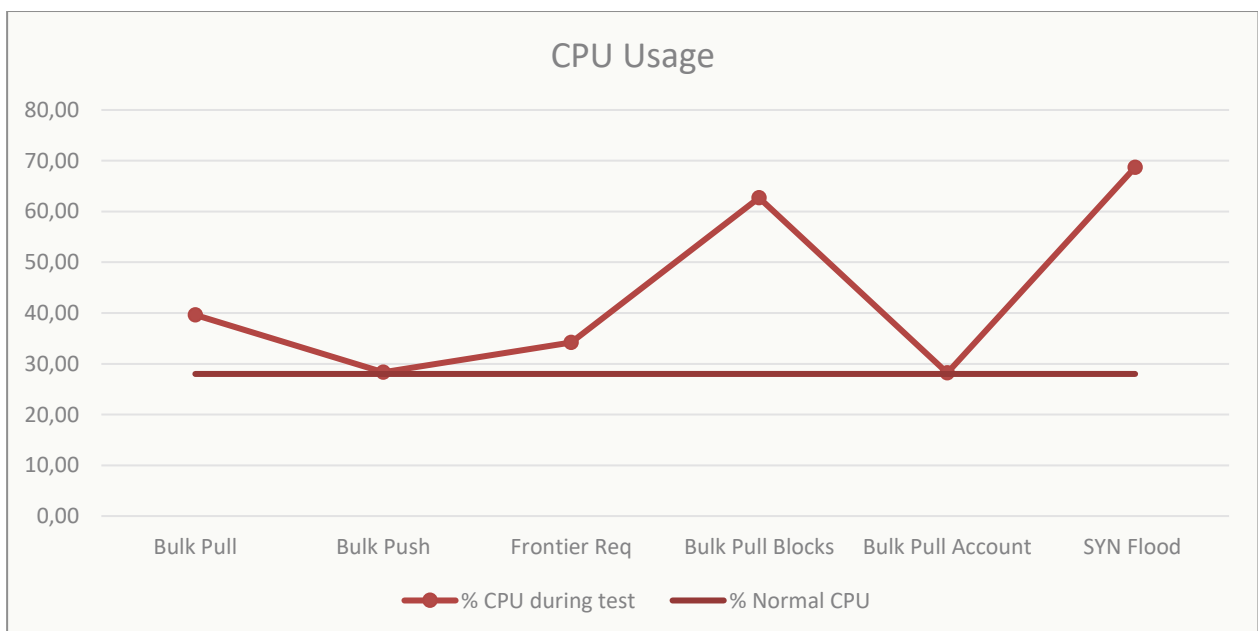
Test	Packets	Kb	Test duration	% CPU	% RAM
Keepalive	101542	13611.362	0:01:00	63.38	64.68
Publish	146604	3197.125	0:01:30	44.19	64.67
Confirm_Req	98570	1931.29	0:01:10	36.00	66.39
Confirm_Ack	95634	27279.602	0:01:00	48.25	66.26
ID Handshake	95634	27279.602	0:01:00	30.36	65.71
Combined tests	99874	13467.931	0:02:00	48.53	65.09

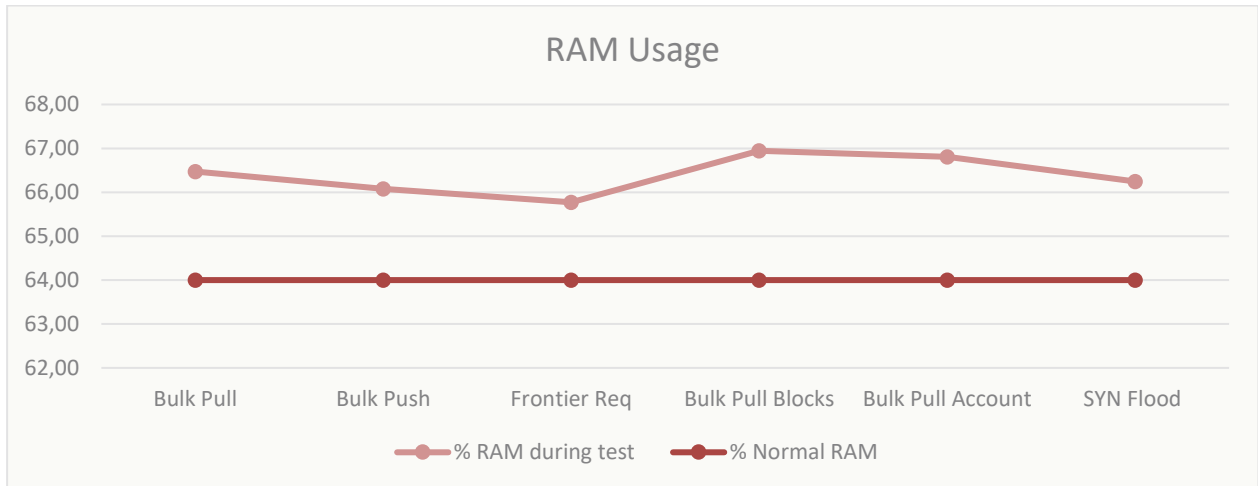


First TCP Simulation



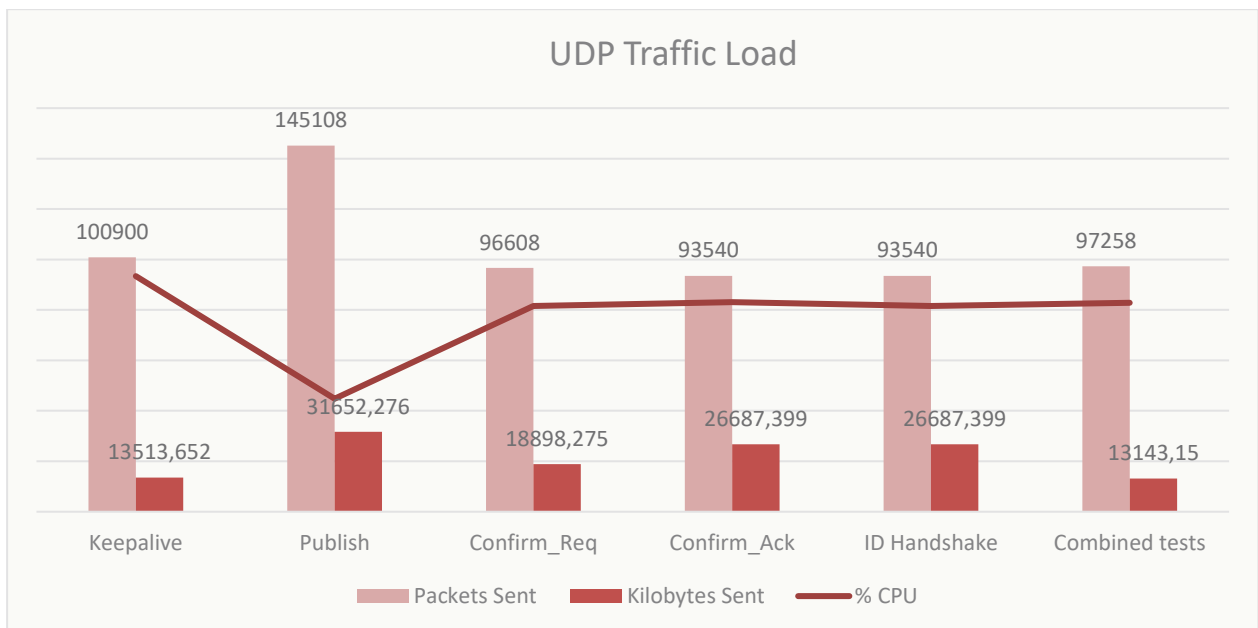
Test	Packets	Bytes	Test duration	% CPU	% RAM
Bulk Pull	9094	144845	0:01:00	39.56	66.47
Bulk Push	8172	16352	0:01:00	28.31	66.08
Frontier Req	8045	63196	0:01:00	34.21	65.77
Bulk Pull Blocks	10989	703604	0:01:30	62.71	66.95
Bulk Pull Account	6651	349648	0:00:30	28.2	66.81
SYN Flood	26625	-	0:01:30	68.71	66.25



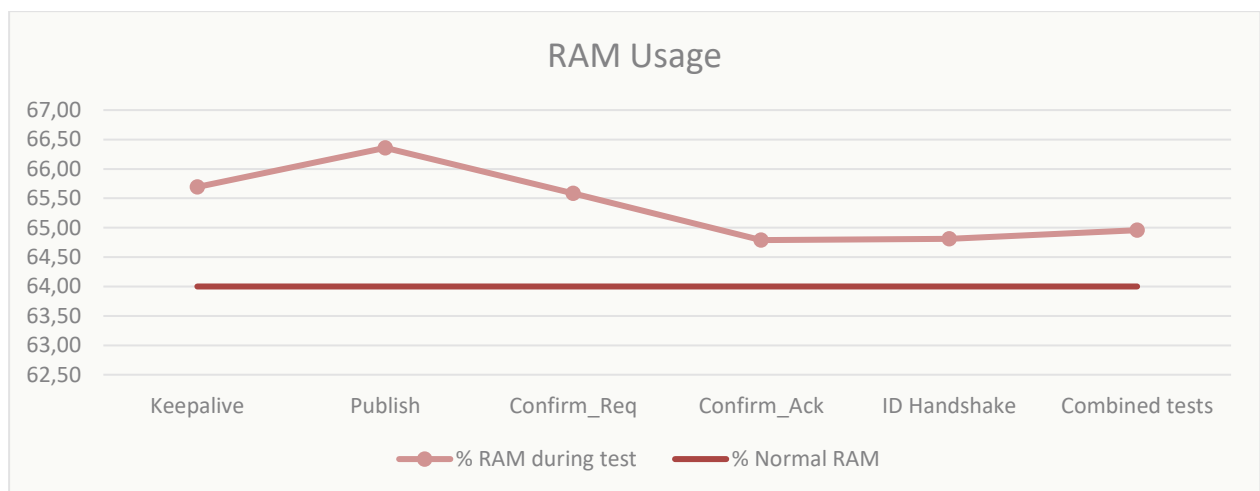
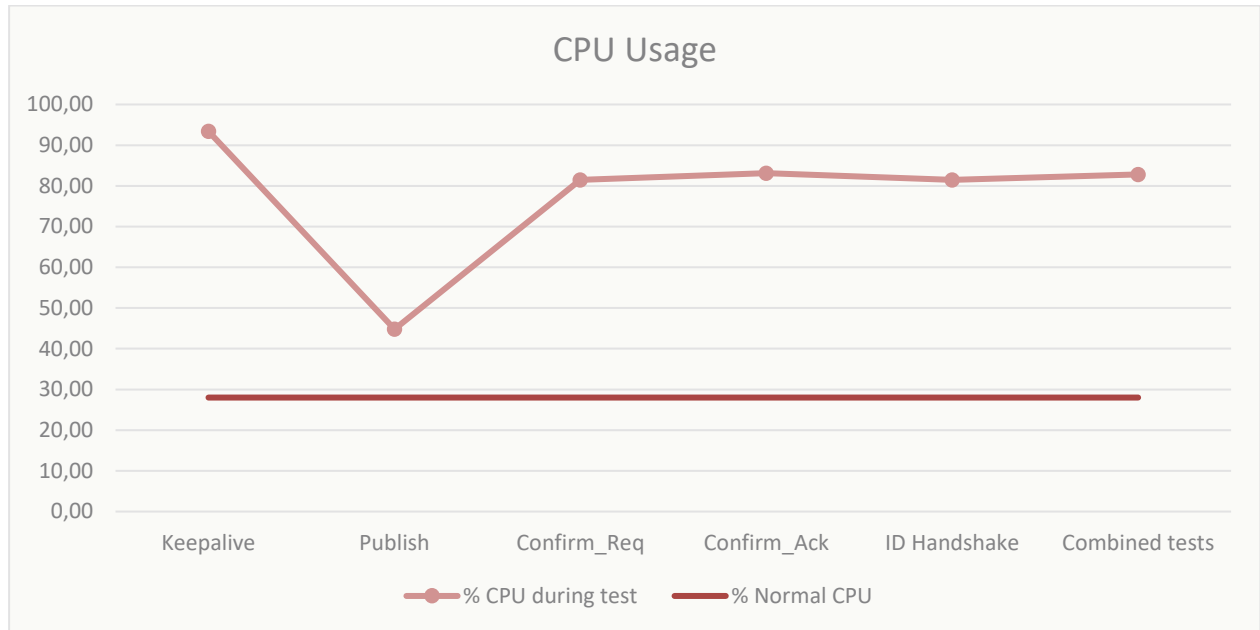


In the following tests, the Proof of Work (PoW) has been removed in order to analyse the behaviour of the service by processing malformed packets that meet the proof of work challenge.

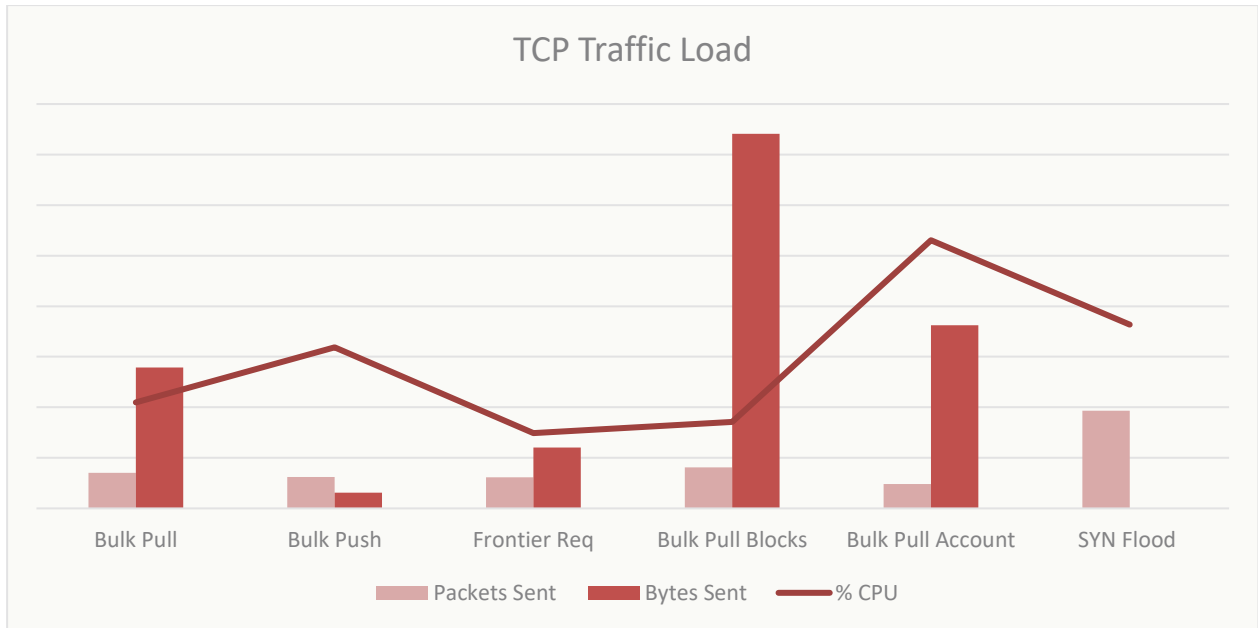
Second UDP Simulation



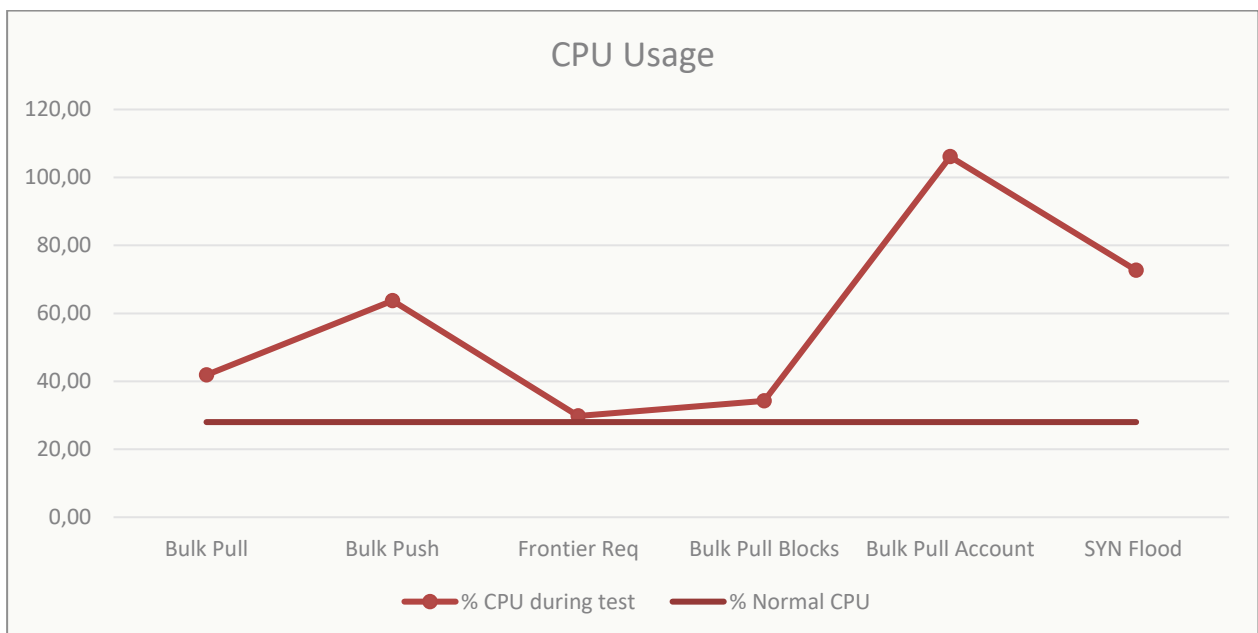
Test	Packets	Kb	Test duration	% CPU	% RAM
Keepalive	100900	13513.652	0:01:00	93.42	65.69
Publish	145108	31652.276	0:01:30	44.75	66.36
Confirm_Req	96608	18898.275	0:01:00	81.45	65.58
Confirm_Ack	93540	26687.399	0:01:00	83.08	64.79
ID Handshake	93540	26687.399	0:01:00	81.52	64.81
Combined tests	97258	13143.15	0:02:00	82.79	64.96

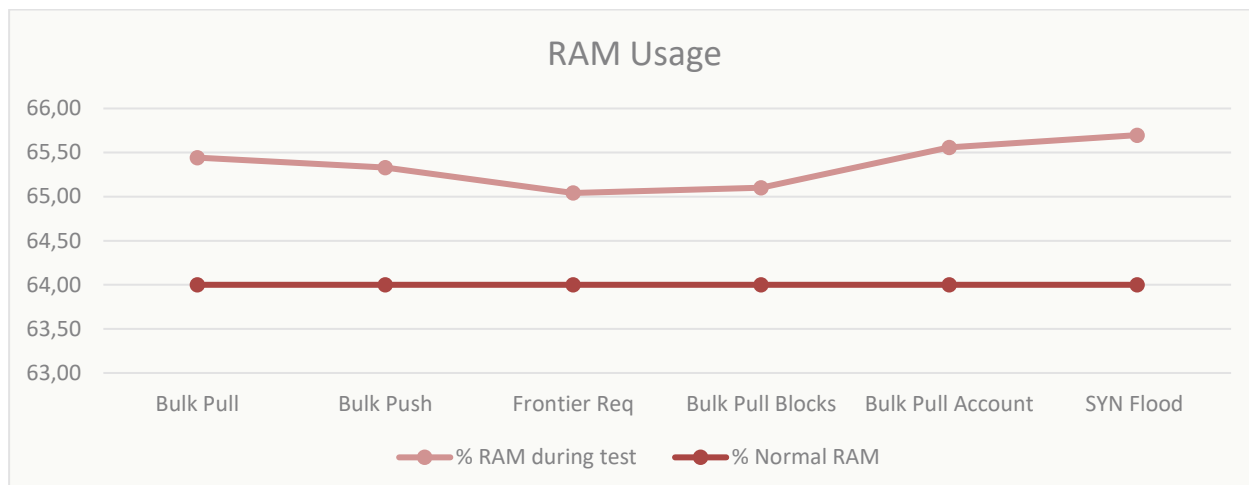


Second TCP Simulation



Test	Packets	Bytes	Test duration	% CPU	% RAM
Bulk Pull	14046	223027	0:01:00	41.93	65.44
Bulk Push	12455	24904	0:01:00	63.68	65.33
Frontier Req	12280	96183	0:01:00	29.79	65.04
Bulk Pull Blocks	16233	1037308	0:01:30	34.18	65.10
Bulk Pull Account	9666	506912	0:00:30	106.13	65.56
SYN Flood	38682	-	0:01:30	72.70	65.70





6.4 Conclusions

After evaluating the results, it can be observed how the challenge of Work Test (PoW) in the packets **properly filters** most of the load while processing malformed packages, even when the service **does not include** any type of protection against distributed denial of service attacks.

Even though the behaviour of the service is quite acceptable, it is always advisable to **establish protection measures** on these types of attacks. This is due to the possibility of identifying and locating the public IP addresses of the representatives through the behaviour of the network and focusing the attacks against them.

Notice that these performed tests only offer a small-scale approach to the impact of a similar attack.

7. Source Code Vulnerabilities

In this section, you can find a detailed analysis of the vulnerabilities encountered during the source code audit.

7.1 Vulnerability Severity

The risk classification has been made on the following 5-value scale:

Critical

- Vulnerabilities that possess the highest impact over the systems, services and/or sensitive information. The existence of this vulnerabilities is dangerous and should be fixed as soon as possible.

High

- Vulnerabilities that could severely compromise the service or the information it manages even if the vulnerability requires expertise to be exploited.

Medium

- Vulnerabilities that on their own can have a limited impact and/or that combined with other vulnerabilities could have a greater impact.

Low

- This vulnerabilities do not suppose a real risk for the systems. Also includes vulnerabilities which are extremely hard to exploit or whose impact on the service is low.

Informative

- It covers various characteristics, information or behaviours that can be considered as inappropriate, without being considered as vulnerabilities by themselves.

7.2 Methodology

All the tests and processes carried out for the achievement of the present project, are included in methodologies and standards recognized and accepted by the international community of software and communications security.

Some examples are **WASC** (Web Application Security Consortium), **MITRE-CWE** that lists the most widespread weaknesses of software, as well as **SEI CERT C/C++** (Software Engineering Institute - Carnegie Mellon University) for the specific taxonomies of C and C++ that are enumerated below:

1. Declarations and Initialization (DCL)
2. Expressions (EXP)
3. Integers (INT)
4. Containers (CTR)
5. Characters and Strings (STR)
6. Memory Management (MEM)
7. Input Output (FIO)
8. Exceptions and Error Handling (ERR)
9. Object Oriented Programming (OOP)
10. Concurrency (CON)
11. Miscellaneous (MSC)

Additionally, other methodologies have been used in addition to the experience of the team, in order to prioritize and perform the tests considered more relevant.

7.3 Coverage

The code audit focuses on the most important fragments of code, which have been previously identified in the scope and those contained in the folder:

- <https://github.com/nanocurrency/nano-node/blob/master/nano/>

Within this folder, the different files have been analyzed in an unequal way, paying special attention to the scope requested by Nano, those related to the classes **rai::active_transactions**, **rai::elections** and the secure folder.

Below is an estimate of the reviewed percentages:

Filename	%
lib/blocks.cpp	100
lib/blocks.hpp	100
lib/config.hpp	100
lib/errors.cpp	40
lib/errors.hpp	100
lib/expected.hpp	100
lib/interface.cpp	40
lib/interface.h	50
lib/numbers.cpp	20
lib/numbers.hpp	100
lib/plat/	60
lib/utility.cpp	60
lib/utility.hpp	100
lib/work.cpp	80
lib/work.hpp	100
node/openclwork.hpp	100
node/peers.cpp	100

Filename	%
node/bootstrap.cpp	100
node/bootstrap.hpp	100
node/cli.cpp	100
node/cli.hpp	100
node/common.cpp	100
node/common.hpp	100
node/lmdb.cpp	100
node/lmdb.hpp	100
node/logging.cpp	100
node/logging.hpp	100
node/nodeconfig.cpp	100
node/nodeconfig.hpp	100
node/node.cpp	100
node/node.hpp	100
node/openclwork.cpp	100
node/xorshift.hpp	100
rai_node/daemon.cpp	80

node/peers.hpp	100
node/plat/	60
node/portmapping.cpp	100
node/portmapping.hpp	100
node/rpc.cpp	90
node/rpc.hpp	90
node/rpc_secure.cpp	60
node/rpc_secure.hpp	70
node/stats.cpp	90
node/stats.hpp	90
node/testing.cpp	70
node/testing.hpp	70
node/voting.cpp	100
node/voting.hpp	100
node/wallet.cpp	100
node/wallet.hpp	100
node/working.hpp	100

rai_node/daemon.hpp	100
rai_node/entry.cpp	100
rai_wallet/entry.cpp	100
rai_wallet/icon.hpp	100
rai_wallet/plat/	60
secure/blockstore.cpp	100
secure/blockstore.hpp	100
secure/common.cpp	100
secure/common.hpp	100
secure/ledger.cpp	100
secure/ledger.hpp	100
secure/plat/	60
secure/utility.cpp	100
secure/utility.hpp	100
secure/versioning.cpp	100
secure/versioning.hpp	100
rai_node/daemon.cpp	80

7.4 Automatic Analysis

During any audit process, in addition to the manual analysis, an **automatic static code analysis** is always performed. In this case, we have used one of the most in-demand tools: *Fortify Static Code Analyzer*.

The main goal is to ensure that all procedural aspects of a code review are covered.

The following table depicts a summary of all issues grouped vertically by Fortify category. For each category, the total number of issues is shown by Fortify Priority Order, including information about the number of audited issues.

Category	Fortify Priority				Total Issues
	Critical	High	Medium	Low	
Buffer Overflow	38	23	0	0	61
Buffer Overflow: Format String	0	3	0	0	3
Buffer Overflow: Off-by-One	1	0	0	0	1

The automatic analysis has detected some issues that have been reviewed and discarded as false positives. This information can be found in Annex B. However, a comprehensive manual review of the code has been carried out.

The table below shows the list of identified vulnerabilities and whether or not they are out of scope or if they have been classified as false positive:

Filename	Occurrences	Status
<code>lmbd/libraries/liblmbd/mdb.c</code>	1	Out of scope
<code>miniupnp/miniupnpc/minissdpc.c</code>	35	Out of scope
<code>miniupnp/miniupnpc/miniupnpc.c</code>	3	Out of scope

7.5 Manual Analysis

During the code audit, different phases have been carried out for the correct understanding of the project as well as the code that implements it.

After first contact, the existence of several third-party libraries for the management of information in memory, UPnP protocol management, as well as other functionalities are observed.

First of all, we proceed to analyze all the public information of the project, from the Whitepaper of the project to the developer's documentation.

Multiple changes are made in the compilation parameters to extract the maximum information at the compile-time. The obtained results will be later evaluated at the debugging and code audit phase.

The last step and one of the most important aspects of the manual analysis is to prepare the fuzzing environment.

Once the previous stages have been carried out, the main code is audited, avoiding the third-party libraries contained therein. We have focused on the classes ***rai::active_transactions*** and ***rai::elections*** in addition to the code located at ***secure*** folder.

All the code was revised several times, dividing this process into several phases to analyze the code based on different approaches. The most important and most suspicious areas were identified for further analysis in debugging mode.

The phases are divided into:

1. Identification of sensitive areas:

- At this point, the zones of the code that handle data, structures and types of data that contain sensitive information, such as weights and voting system, are identified. It is important to also consider the persistence of data, and concurrent access to memory zones.

2. Evaluation of data types and expressions:

- Once the sensitive areas are identified, conditional blocks or loops that control access to those areas are identified. The expressions

and types of data are analysed to identify if it is possible to manipulate the restrictions by modifying the value of the data, and if it has tried to exceed the limits in the loops and cause uncontrolled failures.

3. Application logic:

- Once the operation of the software is understood, situations that could be ambiguous without the appropriate context are identified. From this point, attacks are conducted to test the logic of the application and unexpected behaviour that could be caused.

4. Concurrent access:

- The critical areas are listed and the concurrent accesses are analysed to verify that the flow of access to them is correctly controlled. It is checked if the semaphores are properly established so that they block and unblock their access properly.

5. Dynamic memory management:

- Identify code areas that dynamically manage data and track it. The objective is to identify incorrect memory management or memory zones not released. All this could cause excessive and uncontrolled memory consumption problems.

7.6 List of vulnerabilities

Below we have a complete list of the vulnerabilities detected by Red4Sec, presented and summarized in a way that can be used for risk management and mitigation.

Table of vulnerabilities			
Id.	Vulnerability	Risk	State
1	Improper Validation of Array Index	High	Pending
2	Incorrect Type Conversion or Cast	Informative	Pending
3	Code Styling	Informative	Pending

7.7 Vulnerability details

In this section, we provide the details of each of the detected vulnerabilities indicating the following aspects:

- Category
- Active
- Risk
- Description
- Recommendations

1 - Improper Validation of Array Index

Category	Active	Risk
Insecure library	mdb.c:7118	<div style="display: inline-block; width: 15px; height: 15px; background-color: #e91e63; margin-right: 5px;"></div> High CWE-129

Description:

During the security audit, Red4Sec has detected that some parts of the source code do not have an appropriate validation.

The use of an array has been detected without the proper checking of limits. After an exhaustive analysis of the code it has been observed that it belongs to a third-party library: **Imdb**, which is not in its latest version within the project repository.

The project **rai_blocks** uses the version 0.9.21, while the 0.9.22 is the latest version. This updated version has multiple updates that should be mitigated.

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

↶ base: LMDB_0.9.21 + compare: LMDB_0.9.22

↔ 16 commits
📄 23 files changed
💬 0 commit comments
👤 4 contributors

📅 Commits on Feb 09, 2018		
↕	quanah 0.9.22 engineering	300fc32
↕	quanah ITS#8632 Fix Solaris builds with libldb	52426af
↕	quanah ITS#8632	48c1601
↕	hyc and quanah ITS#8700 fix regression in 0.9.19	a409a75
↕	quanah ITS#8700	e20e307
↕	quanah Fix ITS location	3b1ac04
📅 Commits on Feb 11, 2018		
↕	hyc and quanah ITS#8699 more for cursor_del ITS#8622	35251f6
↕	quanah ITS#8622	2642634
↕	hyc and quanah ITS#8722 fix FIRST_DUP/LAST_DUP cursor bounds check	9852910
↕	hforu and quanah XCURSOR_REFRESH() fixups/cleanup	40daa8e
↕	hforu and quanah Tweak ITS#8722 fix: use XCURSOR_REFRESH()	f6514da
↕	quanah ITS#8722	a351fe0

1 - Improper Validation of Array Index

In the following image, you can see the fix-commit of the vulnerability detected during the audit.



```

6426 6426         rc = MDB_INCOMPATIBLE;
6427 6427         break;
6428 6428     }
6429 6429     if (mc->mc_ki[mc->mc_top] >= NUMKEYS(mc->mc_pg[mc->mc_top])) {
6430 6430         mc->mc_ki[mc->mc_top] = NUMKEYS(mc->mc_pg[mc->mc_top]);
6431 6431         rc = MDB_NOTFOUND;
6432 6432         break;
6433 6433     }
6434 6434     {
6435 6435         MDB_node *leaf = NODEPTR(mc->mc_pg[mc->mc_top], mc->mc_ki[mc->mc_top]);
6436 6436         if (!ISSET(leaf->mn_flags, F_DUPDATA)) {
7000 7000         if (!(m2->mc_flags & C_INITIALIZED)) continue;
7001 7001         if (m2->mc_pg[mc->mc_top] == mp) {
7002 7002             MDB_node *n2 = leaf;
7003 7003             if (m2->mc_ki[mc->mc_top] >= NUMKEYS(mp)) continue;
7004 7004             if (m2->mc_ki[mc->mc_top] != mc->mc_ki[mc->mc_top]) {
7005 7005                 n2 = NODEPTR(mp, m2->mc_ki[mc->mc_top]);
7006 7006                 if (n2->mn_flags & F_SUBDATA) continue;
    
```

The fragment of code affected is in **src/lmdb/libraries/liblmbd/mdb.c**

```

7116         if (!(m2->mc_flags & C_INITIALIZED)) continue;
7117         if (m2->mc_pg[mc->mc_top] == mp) {
7118             MDB_node *n2 = leaf;
7119             if (m2->mc_ki[mc->mc_top] != mc->mc_ki[mc->mc_top]) {
7120                 n2 = NODEPTR(mp, m2->mc_ki[mc->mc_top]);
7121                 if (n2->mn_flags & F_SUBDATA) continue;
    
```


References:

- https://github.com/LMDB/lmdb/compare/LMDB_0.9.21...LMDB_0.9.22
- <https://github.com/LMDB/lmdb/commit/98b2910ee89e9fbc6c2df00d3dd35aea7b86daf>

Recommendations:

- Check the limits properly, managing the correct output in case of error.
- Keep the code updated, especially when it refers to third-party libraries.

2 - Incorrect Type Conversion or Cast

Category	Active	Risk
Insecure Cast or type conversion	node.cpp:2691-2694	 Informative CWE-704

Description:

It has been observed that depending on the compiler or compiler options, it is likely that data types are converted in an inappropriate manner.

The following example belongs to **src/rai/node/node.cpp**

```
2686 rai::election_vote_result rai::election::vote (rai::account rep, uint64_t sequence, rai::block_hash
2687 {
2688     // see republish_vote documentation for an explanation of these rules
2689     auto transaction (node.store.tx_begin_read ());
2690     auto replay (false);
2691     auto supply (node.online_reps.online_stake ());
2692     auto weight (node.ledger.weight (transaction, rep));
2693     auto should_process (false);
2694     if (rai::rai_network == rai::rai_networks::rai_test_network || weight > supply / 1000) // 0.1%
2695     {
2696         unsigned int cooldown;
2697         if (weight < supply / 100) // 0.1% to 1%
2698         {
2699             cooldown = 15;
2700         }
2701         else if (weight < supply / 20) // 1% to 5%
2702         {
2703             cooldown = 5;
2704         }
2705         else // 5% or above
2706         {
2707             cooldown = 1;
2708         }
2709         auto last_vote_it (last_votes.find (rep));
2710         if (last_vote_it == last_votes.end ())
2711         {
2712             should_process = true;
2713         }
2714         else
```

Notice that both supply and weight are of the type **rai::uint128_t** while the value 1000 is taken as signed int.

2 - Incorrect Type Conversion or Cast

To carry out the comparison, supply is first divided by 1000. At this point, the default action is to convert the variable of smaller capacity, to one of greater capacity, and perform the operation, which converts unsigned int to *uint128_t*, losing the sign.

Since it is not specified, the decision on the comparison depends on the compiler's policy, being able to:

1. Prioritize maintaining the sign with respect to the value.
2. Convert supply into signed int and lose its value.

When making the comparison, it is possible to extend this conversion to the variable weight and this could cause unexpected situations and with it execute the code of the conditional, when in reality it should not.

References:

- <https://github.com/nanocurrency/nano-node/blob/1caec639ad8cf66a3dc90092e280a043e5dbb86b/rai/node/node.cpp#L2985>
- <https://github.com/nanocurrency/nano-node/blob/1caec639ad8cf66a3dc90092e280a043e5dbb86b/rai/node/node.cpp#L2988>

Recommendations:

- It is recommended to carry out an exhaustive verification of all the errors that the vulnerability's code object could produce.
- Conduct the conversions explicitly

3 - Code Styling

Category

Active

Risk

Bad practices

Nano Source Code

 **Informative**
CWE-398**Description:**

It has been possible to verify that, in spite of good quality code, there is a lack of order and structure that makes reading and analyzing the code difficult.

This is a very common bad practice, especially in these types of projects that are continually changing and improving. This is not a vulnerability in itself, but it helps to improve the code and reduces the appearance of new vulnerabilities.

As a reference, it is always recommendable to apply some coding style/good practices that can be found in multiple standards such as:

- "Google C++ Style Guide" (<https://google.github.io/styleguide/cppguide.html>).
- "ISOCPP Core Guidelines" (<https://isocpp.org/wiki/faq/coding-standards>).

These references are very useful to improve quality software. Some of those practices are common and a popular accepted way to develop software.

In this project, the team found some issues related with the coding style that should be taken into account. It's highly recommended to follow the coding style good practices from any formal standard.

For example:

"A very common case is to have a pair of files called, e.g., foo_bar.h and foo_bar.cc, defining a class called FooBar."

References:

- https://google.github.io/styleguide/cppguide.html#File_Names

3 - Code Styling

Recommendations:

- Implement best practices to optimize code performance.
- Follow code style/good practices during the whole development process.

8. Annexes

In the annexes, information referenced in the document is included as well as information related to the security review performed.

The information found in the annexes mainly includes:

- List of conducted tests.

Annex A List of conducted tests

The following states have been defined and used during the execution of the review plan, to manage the revision process.

Test	State
The test has been scheduled but has not yet started.	(P) Pending
The execution of the tests has been suspended since none of the necessary elements for its realization exists, given its low priority or being outside the scope of the audit.	(S) Suspended
The test has been performed during the test battery.	(A) Accomplished
The test has been excluded after being previously agreed with the client.	(D) Deleted

The final status of the agreed tests in the Revision Plan, once finished, is as follows:

Conducted Tests	State	Observations
INPUT VALIDATION		
Cross-Site Scripting	S	
XPath Injection	S	
LDAP Injection	S	
Buffer Overflow	A	
Memory Corruption	A	
SOURCE CODE DESIGN		
Insecure field scope	A	
Insecure method scope	A	
Insecure class modifiers	A	
Unused external references	A	

Redundant code	A	
INFORMATION LEAKAGE AND IMPROPER ERROR HANDLING		
Unhandled exception	A	
Routine return value usage	A	
NULL Pointer dereference	A	
Insecure logging	A	
DIRECT OBJECT REFERENCE		
Direct reference to database data	A	
Direct reference to filesystem	A	
Direct reference to memory	A	
RESOURCE USAGE		
Insecure file modifying	A	
Insecure file deletion	A	
Race conditions	A	
Memory leak	A	
Unsafe process creation	A	
CRYPTOGRAPHY IMPLEMENTATIONS		
Key exchange algorithms	A	
Cryptographic primitives	A	
Account Management and transfers	A	
Wallet (Seed Generation, Key Derivation, Encryption)	A	
Cryptographic Complements	A	
BEST PRACTICES VIOLATION		
Insecure memory pointer usage	A	
NULL Pointer dereference	A	
Pointer arithmetic	A	
Variable aliasing	A	
Unsafe variable initialization	A	
Missing comments and source code documentation	A	
WEAK SESSION MANAGEMENT		
Not checking for valid sessions upon HTTP request	S	
Not invalidating session upon an error occurring	S	
Not issuing a new session upon successful authentication	S	
Passing cookies over non-SSL connections (No Secure Flag)	S	



Invest in Security, invest in your future