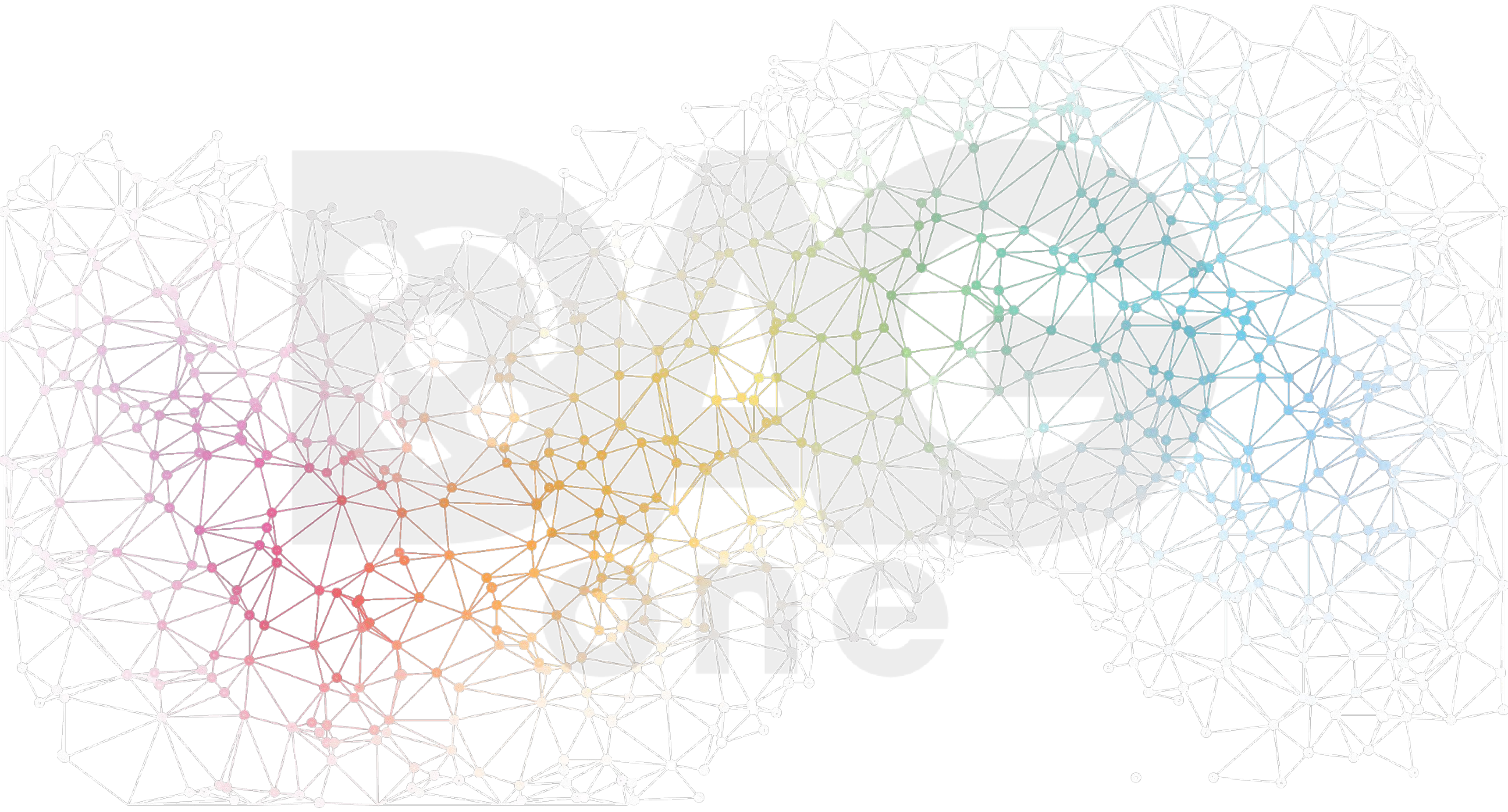


PERLIN:WAVELET

Distributed Ledger Platform

SECURITY AUDIT REPORT

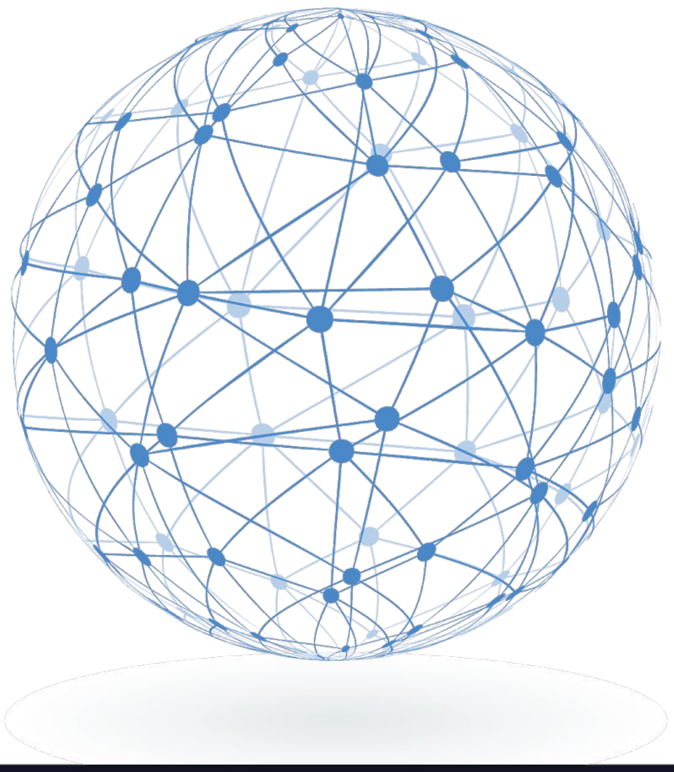
Prepared by:
DAG-ONE LLC

July 1, 2019

Table of Contents

Introduction

Perlin originally bootstrapped on top of the Avalanche consensus protocol. Since its inception, however, Perlin identified several scalability, practicality and safety concerns inherent to the Avalanche protocol. Perlin immediately set about solving these shortcomings. In September 2018, Perlin released its solution: Wavelet. Included in the Wavelet rollout was an Overlay Routing system, a leaderless proof-of-stake Sybil resistance mechanism, a rewards system for honest nodes and penalties for misbehaving ones, a novel transaction ordering system to ensure smart contract compatibility, among many other improvements. *Wavelet* is a distributed ledger system with smart contracts, written in Go, built for developing decentralized applications and a digital currency. Perlin has engaged Dag-One to conduct a security audit of Wavelet.



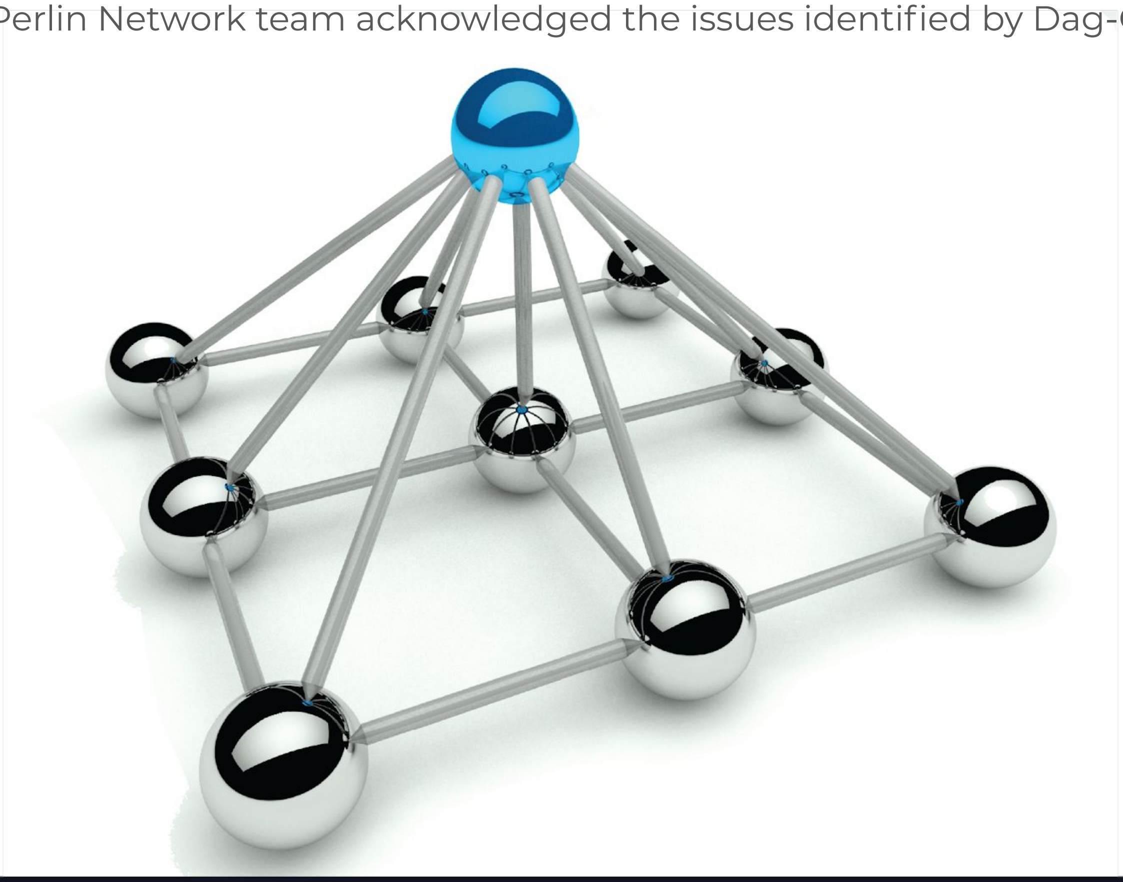
Disclaimer: *Dag-One does not endorse or approve products and/or services used or developed by third parties. This audit was conducted for the exclusive use of the client and is for discussion purposes only. This report does not constitute a public endorsement of the reliability, effectiveness or usefulness of any product or service.*

Executive Summary

The audit is a reflection of the code in the Perlin Network's Wavelet repository on Github. It took place over the course of six weeks from May 15, 2019, to June 28, 2019. The evaluation was conducted to identify vulnerabilities and defects in the code that could cause problems for the users of Wavelet, particularly within the context of Perlin Network's usage of it as the basis for a sustainable cryptocurrency and decentralized application platform. During the audit, the following methodologies were used: fuzz testing, heuristics, failure mode and effect testing, and code review. In addition Dag-One performed benchmarking on Wavelet to determine performance parameters and critical paths which could be improved.

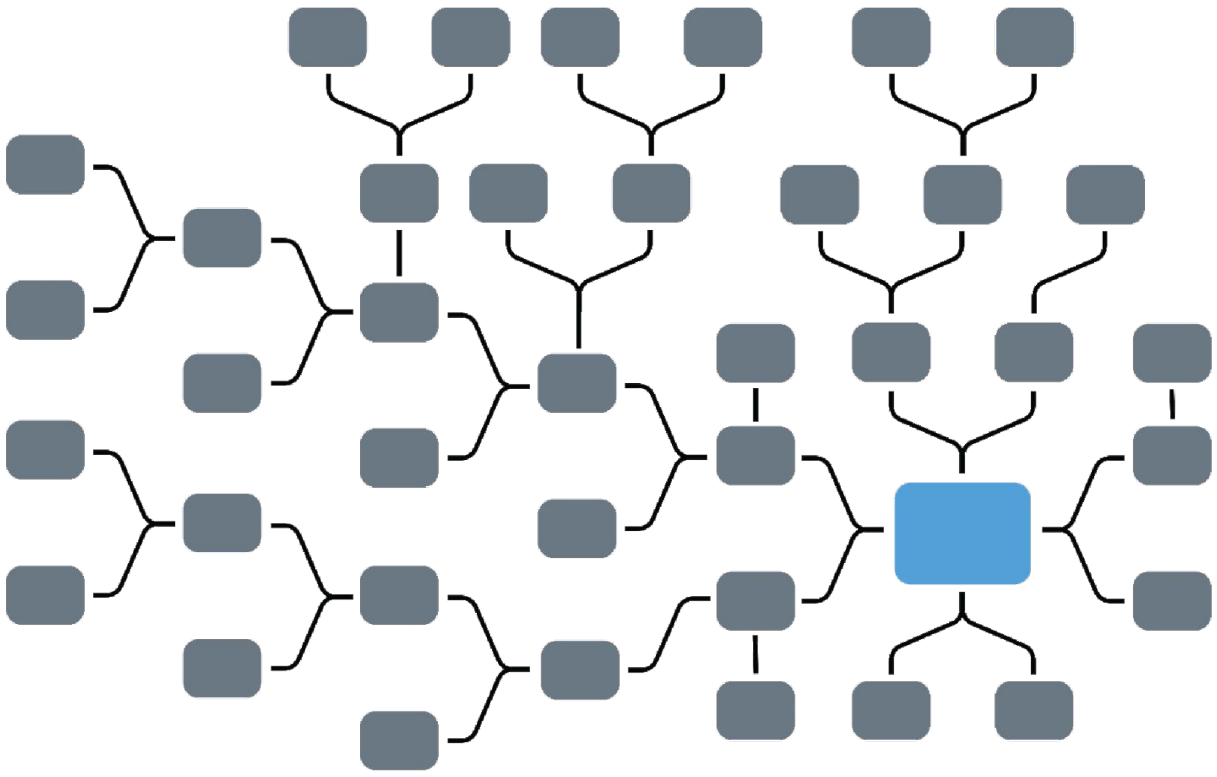
Additional information about the methodologies used can all be found in their respective sections.

Altogether there were five issues noted, one of which had a critical severity, two had a medium severity level, and two with a low severity level. The Perlin Network team acknowledged the issues identified by Dag-One.



List of Repositories Audited

Repository Name	Wavelet
Repository URL	https://github.com/perlin-network/wavelet
Commit ID	`d1b2fc73360f087e73c82a4f52c0a3e1ae59e92c`



Audit Methodology

FUZZ TESTING

The purpose of fuzz testing is to discover coding errors and security loopholes in a target system by deploying an automated or semi-automated program (a "fuzzer") to provide invalid, unexpected, or random data to the target system. In order for the fuzzer to be effective it must be capable of generating inputs that are valid enough to be accepted by the target system's parser. The goal is to identify unexpected behaviors caused by the fuzzing process that expose the target system to vulnerabilities.

Steps Taken:

1. Define the target system
    a. Fuzzing was done over the exposed interfaces; and
    b. Fuzzing was done over the Transaction Parser
2. Identify the inputs to be used
3. Generate random fuzzed data
4. Deploy the fuzzed data to the target system
5. Monitor system behavior
6. Log defects

Audit Methodology cont.

HEURISTICS

A heuristic-based evaluation system is one which uses a specific set of principles or usability guidelines to evaluate the effectiveness of a particular implementation. While the principles or guidelines may themselves be subjective, whether or not the target system follows the given guidelines is objective.

Steps Taken:
We have used two outside heuristics evaluators to review the target system:

1. Evaluate reputation of dependencies
2. Static Analysis

Wavelet has 48 dependencies including all transitive dependencies. The majority of these dependencies are in good health, of good quality, and used by many larger projects.

Audit Methodology cont.

FAILURE MODE AND EFFECT TESTING

A failure mode and effect evaluation system is one in which possible failure modes are run in a simulated environment and changes applied under specific conditions to cause failures to occur and observe the effects. This includes not only the first-order effects but second-order and higher as well. The goal with failure mode and effect evaluation is to determine how a system operates near, at, or beyond its designed limits.

Steps Taken:

The process for failure mode and effect evaluation involves:

1. Determine possible or likely conditions in which the software may fail;
2. Attempt to create that environment;
3. Apply changes to stress the software further, until the test ends or no more change can be observed in the software.

The scenarios Wavelet were tested under include the following classes of nodes:

1. *Good*: The type of node which includes the Genesis wallet owner, operates as a legitimate node, and pays 1000000 PERL to the addresses of all its immediately connected peers every 60s;
2. *Reflector*: The type of node which acts as a TCP reflector, accepting TCP connections on the Wavelet port and acting as a proxy to other kinds of nodes;

Audit Methodology cont.

FAILURE MODE AND EFFECT TESTING CONTINUED

3. *Spammer*: The type of node which attempts to generate a lot of legitimate network traffic and spurious ledger entries by paying 1 PERL as fast as possible to random 256-bit addresses;
4. *Splitter*: This type of node always uses the same S/Kademlia private key across multiple running instances with different IP address;
5. *Tarpit*: This type of node listens on the Wavelet port and connects to all other kinds of nodes proactively, but only receives data from its socket at 1 byte every 5 seconds on each of those connections;
6. *Genesis*: This type of node is similar to the Good node instance 0; it acts as the genesis wallet node and is controlled by other kinds of nodes for specific tests, but is otherwise passive;
7. *Staker*: This type of node works with the *Genesis* node to arrange obtaining a partial balance for all the PERLs in existence, then staking those PERLs;
8. *Lamer*: This type of node acts like a *Staker* node, but never votes its staked weight

Using different combinations of these classes of nodes we are able to create scenarios in which it is conceivable that Wavelet may fail. The specific scenarios tested in this audit were:

1. *Simple*: Which consists of only a few *Good* nodes, as a baseline to establish that the software does works in the best case scenario;
2. *Spam*: Which consists of a few *Good* nodes and many *Spammer* nodes to attempt to create issues with the ledgers on any of the nodes within the network;

Audit Methodology cont.

FAILURE MODE AND EFFECT TESTING CONTINUED

3. *Reflection*: Which consists of a few *Good* nodes and many *Reflector* nodes in an attempt to see if having extremely lightweight nodes could be used to manipulate the network conditions and fragment the network at low cost;
4. *Split*: Which consists of a few *Good* nodes and many *Splitter* nodes in an attempt to cause honest nodes to be unable to effectively communicate with other nodes;
5. *Big*: A few *Good* nodes, many *Splitter* nodes, many *Tarpit* nodes, many *Reflector* nodes, and a few *Spammer* nodes in a single network to attempt to stress the software by combining all these factors simultaneously
6. *Staking*: A single *Genesis* node, one *Staker* node, one *Lamer* node, and a few *Good* nodes to attempt to perform transactions when less than the minimum amount of weight is available to validate them

Wavelet operated within expected parameters for all of these scenarios.

Audit Methodology cont.

CODE REVIEW

Unlike fuzzing or heuristics-based testing, whose results are objective because their source is 100% auditable, a code review is conducted by one or more humans who conduct a manual review of the target system with the goal of improving the code's quality, efficiency or implementations. Because of the subjective nature of a code review it is important that the steps taken during the code review be carefully documented and recorded. Heuristics and fuzzing provide guidance to those conducting a code review.

Steps Taken:

1. Read and understand the code of the target system;
2. Look for logical/semantic faults;
3. Consider side channel attacks;
4. Recommend improvements;
5. Document all steps taken.

SECURITY IMPACT LEVELS

We have generated the following security levels based on the needs of the project and the Common Vulnerability Scoring System by the National Institute of Standards and Technology (NIST).

Security Level	Description
Critical/High	The vulnerability may result in a total loss of system integrity, resulting in all resources within the impacted component being vulnerable to the attacker. The result is that the attacker is able to fully deny access to resources in the impacted component; this loss is either sustained (while the attacker continues to deliver the attack) or persistent (the condition persists even after the attack has been completed).
Medium	The vulnerability may result in a partial loss of system integrity that nevertheless provides the attacker with access to essential resources, which could nevertheless cause an architectural failure. The attacker has the ability to deny some availability, but the loss of availability presents a direct, serious consequence to the impacted component (e.g., the attacker cannot disrupt existing connections, but can prevent new connections; the attacker can repeatedly exploit a vulnerability that, in each instance of a successful attack, leaks a only small amount of memory, but after repeated exploitation causes a service to become completely unavailable).
Low	The vulnerability may result in reduced performance or interruptions in resource availability. Even if repeated exploitation of the vulnerability is possible, the attacker does not have the ability to completely deny service to legitimate users. The resources in the impacted component are either partially available all of the time, or fully available only some of the time, but overall there is no direct, serious consequence to the impacted component.
Informational	No specific vulnerability has been identified, but a best practice has not been followed which may cause user confusion or make user adoption more difficult.

Vulnerabilities

Issue 1:

Description	Ledger ordering error allows for minting new coins
Severity	Critical
Status	Resolved
Information	Due to the way transactions are ordered, sending PERL from one's own account to the same account results in the received amount being added to the the account's balance before the sending is deducted.
Recommendation	Fix the ordering and add a unit and integration test around this behavior.

Issue 2:

Description	Unbounded allocation
Severity	Medium
Status	Resolved
Information	A `UInt32` is read from the network, and a buffer of that size allocated without validation that the buffer size is "reasonable".
Recommendation	Limit the size of this buffer to 128KiB for most kinds of transactions and 4MiB for contract transactions.

Issue 3:

Description	Dependencies at risk
Severity	Medium
Status	In Progress
Information	Wavelet has a large total number of dependencies, some of these dependencies appear to have very little activity and may be at risk of becoming unmaintained.
Recommendation	Vendor dependencies as needed and attempt to minimize the number of “fringe” dependencies that other, larger projects are not using.

Issue 4:

Description	Denial of Service
Severity	Low
Status	In Progress
Information	Building a tree of transactions which depend on transactions which have been pruned but not synced on some nodes lead to those nodes being unable to get the updated version of the ledger.
Recommendation	Improve ledger synchronization mechanism.

Issue 5:

Description	New wallet private keys are created world-readable
Severity	Low
Status	In Progress
Information	The development tool "wavelet-cli" creates wallet files containing private keys with mode 0755, which is world readable.
Recommendation	Adjust mode for new files to 0600.

Note 1:

Benchmark Results	
Methodology	A cluster of 20 nodes was created using Kubernetes on Amazon AWS EKS. Each node was an EC2 instance of type m5.large which has 2 vCPU and 8GiB RAM.
Results	17,200 transactions per second

***Disclaimer:** Dag-One does not endorse or approve products and/or services used or developed by third parties. This audit was conducted for the exclusive use of the client and is for discussion purposes only. This report does not constitute a public endorsement of the reliability, effectiveness or usefulness of any product or service.*

CONCLUSION

Perlin's Wavelet is a high quality Distributed Ledger written in the Go Language. It provides an environment for decentralized application developers to host unstoppable applications. It builds on the work of the other Perlin projects of Noise, a peer-to-peer communications library, and Life, a WebAssembly execution environment.

Wavelet is still under active development but its current implementation exhibits very quality code with few exceptions. The layering between Wavelet and the other components it builds on are generally excellent, resulting in a project that can be maintained going forward and extended for the project's needs for the foreseeable future.

During the audit of the Wavelet code base, we uncovered only a single major issue, which was quickly identified and fixed by the Wavelet development team, as well as a small number of minor issues. The overall results are excellent and this can be largely attributed to the use of a relatively safe high-level language, re-use of audited libraries, and careful planning on the part of the developers.

APPENDIX A

EVIDENCE OF FINDINGS

Issue 1: Improper ordering of ledger operations

Figure 1: Issue 1: Payment to self mints coins

```
»»» find 10d7244186b504b466ea8c9795f12bcf896cc47746b55a93ea4a3b826f79ed1f
1:58AM INF Account:
10d7244186b504b466ea8c9795f12bcf896cc47746b55a93ea4a3b826f79ed1f balance: 10
is_contract: false
nonce: 0 num_pages: 0 reward: 0 stake: 0
»»» pay 10d7244186b504b466ea8c9795f12bcf896cc47746b55a93ea4a3b826f79ed1f 10
1:58AM INF Success! Your payment transaction ID:
4386ec3ab033c18412c2d9520c22208c8834117373275a6140da4187fb3528a6
»»» find 10d7244186b504b466ea8c9795f12bcf896cc47746b55a93ea4a3b826f79ed1f
1:58AM INF Account:
10d7244186b504b466ea8c9795f12bcf896cc47746b55a93ea4a3b826f79ed1f balance: 20
is_contract: false
nonce: 76 num_pages: 0 reward: 0 stake: 0
```

EVIDENCE OF FINDINGS CONT.

Issue 2: Unbounded allocation

Figure 2: Issue 2: Example of unbounded allocation

```
if _, err := io.ReadFull(r, b[:4]); err != nil {
        return tx, errors.Wrap(err, "[...]")
}

tx.FuncParams = make([]byte, binary.LittleEndian.Uint32(b[:4]))
```

Figure 3: Issue 2: Example of mitigation

```
if _, err := io.ReadFull(r, b[:4]); err != nil {
        return tx, errors.Wrap(err, "[...]")
}
size := binary.LittleEndian.Uint32(b[:4])
if (size > 128 * 1024) {
        return tx, errors.new("[...]"])
}
tx.FuncParams = make([]byte, size)
```

EVIDENCE OF FINDINGS CONT.

Note 1: Sybil Resistance

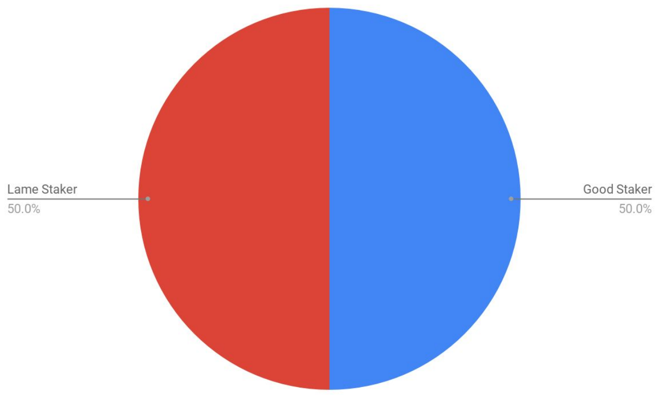


Figure 4: Transactions could still take place even though 50% of the network stake weight was refusing to vote, advancing round for all participating nodes

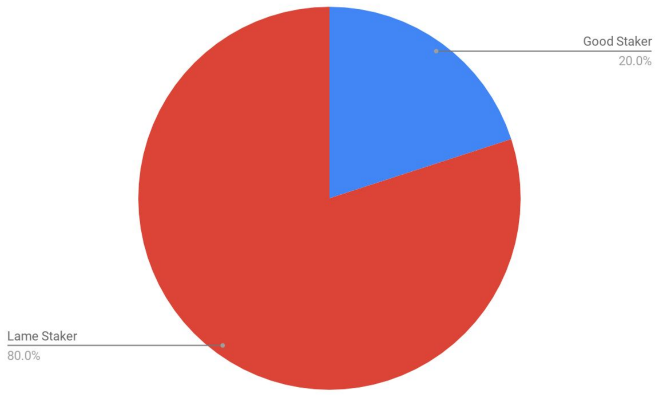


Figure 5: Transactions could still take place even though 80% of the network stake weight was refusing to vote, advancing round for all participating nodes, and requiring more rounds of voting

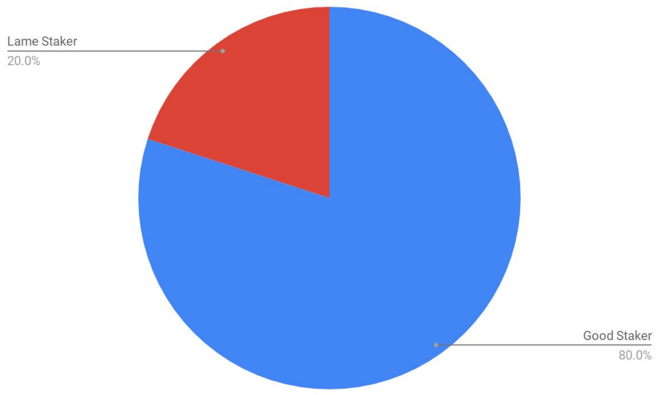


Figure 6: Transactions could still take place with 20% of the stake weight refusing to vote, advancing round for all participating nodes