Stacks Chain Process, Clarity VM, SIP 001 and SIP 007 Cryptography Review

Blockstack

November 23, 2020 – Version 1.0

Prepared for

Blockstack PBC

Prepared by

NCC Group Cryptography Services

Executive Summary

Synopsis

During February 2020, Blockstack engaged NCC Group Cryptography Services to perform a security-focused review consisting of 10 consultant days with two consultants to review of the Stacks 2.0 implementation. Two consultants were staffed on the project over the course of 5 days. The scope of the project included the blockchain implementation (excluding P2P aspects), the authenticated index data structure (MARF) and its integration with the underlying store, and finally the Clarity VM implementation.

During the fall of 2020, Blockstack engaged NCC Group Cryptography Services to perform a security-focused review of `blockstack-core`, with focus on what's described in SIP-001 and SIP-007. In SIP-001, key aspects of the Proof-of-Burn (PoB) concepts are laid out, such as leader election via sortition, interaction with the burnchain and burning pools. In SIP-007, key aspects of the Proof-of-Transfer (PoX) blockchain are described, including reward handling and delegation. The scope was the sortition implementation, interaction with Bitcoin and Bitcoin fork handling. On the kick-off call, Blockstack suggested to look at the Clarity contract governing the PoX rewards as well. One NCC Group consultant worked on the project for 5 days. Blockstack answered consultant's questions in Slack. Testing was performed by manual source code review and by running unit tests.

Scope

Blockchain audit

The primary focus of testing was on code paths around:

- Transaction validation: signature validation, transaction malleability, replay
- Block validation: burnchain registration and commitment, block/microblock binding, block header field validation

More generally, Stacks 2.0 being a from-scratch blockchain implementation, some of the important attack vectors include:

- Netsplit via block hash poisoning
- Netsplit due to multiple client implementations
- Transaction malleability issues
- Integer underflow/overflow
- Merkle tree implementation issues
- Storage exhaustion in block or transaction processing
- CPU exhaustion in block or transaction processing

MARF audit

Regarding the MARF review, NCC Group reviewed code paths with focus on the following vectors:

- Construction of inappropriate proofs, e.g. provers omitting child nodes
- Incorrect validation of proofs by verifiers
- Denial of service vectors in the construction and validation of proofs

NCC Group performed a mix of static and dynamic analysis of MARF to identify and exploit failures to authenticate data indexed by MARF; the latter by expanding some of the existing unit test cases with invalid or unexpected input.

In Phase 2, NCC Group's evaluation included:

- **Sortition implementation**: Proof-of-Burn leader election powered by a Verifiable Random Function (VRF), as described in SIP-001
- **Bitcoin interactions:** Stacks blockchain interactions with the burnchain, including burnchain forks edge case handling
- **PoX Clarity Smart Contract:** Key elements of SIP-007's logic are handled inside the Clarity PoX contract and `blockchain-core` calls out to the contract.

Testing Methodology and Key Findings

Phase 1 findings included the following:

- **Proof Verification May Not Check the Root Hash:** An attacker may conceal the actual map value of a map key for a given root hash in a proof.
- **Discrepancy Between SIP 005 and Implementation:** If a reimplementation of the Stacks blockchain client follows SIP 005, it risks being forked off the main network.
- **Block and Transaction Encoding Tolerates Arbitrary Suffix:** If, in the P2P layer, messages are not deserialized and then serialized again before being stored or forwarded to other nodes, DoS concerns may arise.

Testing methodology consisted of mapping various aspects of SIP-001 and SIP-007 to the code base, and while doing so, reviewing for:

- Usage of panicking functions
- Integer underflow/overflow issues
- Logical and edge-case handling problems

No serious issues on those tracks have been found. The Phase 2 review noted the following:

- Reward Address Duplication and SIP-007 Inaccuracy: When it comes to PoX reward address computation, SIP-007 diverges from the implementation, resulting in misinformation on how rewards work and complicate a possible re-implementation.
- `SEED_NORM` Calculation Does Not Follow SIP 001: SIP 001 readers will be misinformed on how the `SEED_NORM` quantity (used to pick the sortition winner) is computed. Since SIP-001 is descriptive in nature, it is not likely that such a discrepancy would creep into an actual re-implementation.

Limitations

The Clarity contract came at the end and while interactions with the contract (from `blockstack-core`) were reviewed, there was not sufficient time to review the contract itself as a standalone piece as the review was tightly scoped.

Recommendation

Consider performing a separate independent review of the PoX Clarity Contract, with focus on logical issues such as unintended edge-case handling and execution correctness.

Dashboard

Finding Breakdown

Critical issues	0	
High issues	1	🟥
Medium issues	2	🟧🟧
Low issues	2	🟨🟨
Informational issues	2	⬜⬜
Total issues	**7**	

Category Breakdown

Cryptography	3	🟥🟨⬜
Denial of Service	2	🟧🟧
Other	2	🟨⬜

Component Breakdown

Clarity VM	2	🟧🟧
General	1	⬜
MARF	1	🟥
SIP 001	1	⬜
SIP-007	1	🟨
Transaction validation	1	🟨

Key

🟥 Critical 🟥 High 🟧 Medium 🟨 Low ⬜ Informational

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix A on page 16.

Title	Status	ID	Risk
Proof Verification May Not Check the Root Hash	New	001	High
Unbounded Recursion in Contract Parser Leads To Crash	New	004	Medium
Denial of Service via ClarityVM Process Thrashing	New	005	Medium
Discrepancies Between SIP 005 and Implementation	New	002	Low
Reward Address Duplication and SIP-007 Inaccuracy	Reported	007	Low
Block and Transaction Encoding Tolerates Arbitrary Suffix	New	003	Informational
`SEED_NORM` Calculation Does Not Follow SIP 001	Reported	006	Informational

Finding Details

Finding	**Proof Verification May Not Check the Root Hash**
Risk	**High** Impact: High, Exploitability: Medium
Identifier	NCC-BLSK002-001
Status	New
Category	Cryptography
Component	MARF
Location	proofs.rs:921
Impact	An attacker may conceal the actual map value of a map key for a given root hash in a proof. The impact and exploitability ratings are contingent on a number of factors including the nature of the data being stored and type of smart contract; it is envisaged that these may vary from low to high.

Description

A Merklized Adaptive Radix Forest[1] (MARF) is an authenticated index data structure for encoding a cryptographic commitment to the blockchain state. The MARF structure is part of the consensus logic and gives light clients the ability to verify that a particular key has a particular value, given the MARF's cryptographic hash.

The `chainstate::stacks::index::proofs` module implements a `verify_proof()`[2] method for the `TrieMerkleProof` structure to verify that a given a map value and the root hash from which a proof was generated is consistent with the passed root hash value.

At the aforementioned code path of the `verify_proof()` method, the application has validated that the given map value matches the value in the given proof, among other controls. However, it has not checked whether the given root hash value matches the given proof's root hash value before returning that it has successfully verified the proof. This is illustrated in the code snippet below:

```
trace!("shunt proof head hash: {:?}", &trie_hash);

        i += 1;
        if i >= proof.len() {
            // done -- no further shunts
            return true;
        }
```

If an attacker can generate a proof that matches the above condition, then they may be able to lie about the actual map value. NCC Group was able to hide the most up to date map value in a unit test case by generating a proof for the queried value for an old block (the queried value was correct in that old block but was changed in later blocks) and presenting this proof to the verifying function. The unit test case is documented in Appendix D on page 21.

Recommendation

Ensure that the aforementioned code compares the provided root hash value with the provided proof's root hash value and returns false if they do not match.

[1] https://github.com/blockstack/blockstack-core/blob/develop/sip/sip-004-materialized-view.md
[2] https://github.com/blockstack/blockstack-core/blob/b2e456cef16e943cd5cbf2c0173eabf152a11659/src/chainstate/stacks/index/proofs.rs#L855

Finding **Unbounded Recursion in Contract Parser Leads To Crash**

Risk **Medium** Impact: Medium, Exploitability: Medium

Identifier NCC-BLSK002-004

Status New

Category Denial of Service

Component Clarity VM

Location
- ExpressionIdentifier
- DefinitionSorter
- TraitsResolver
- SugarExpander

Impact An attacker can send poisonous transactions and crash arbitrary nodes on the network.

Description Processing a transaction with a new smart contract payload involves building an AST tree out of the user-supplied contract code. The `vm::ast::build_ast` function performs several passes over the supplied code and gradually builds the `ContractAST` struct:

```
pub fn build_ast(contract_identifier: &QualifiedContractIdentifier, source_code:
➜ &str) -> ParseResult<ContractAST> {
    let pre_expressions = parser::parse(source_code)?;
    let mut contract_ast = ContractAST::new(contract_identifier.clone(),
    ➜ pre_expressions);
    ExpressionIdentifier::run_pass(&mut contract_ast)?;
    DefinitionSorter::run_pass(&mut contract_ast)?;
    TraitsResolver::run_pass(&mut contract_ast)?;
    SugarExpander::run_pass(&mut contract_ast)?;
    Ok(contract_ast)
}
```

All four `run_pass` invocations potentially lead to unbounded recursion. A malicious network participant may construct contract code which will crash nodes, depending on their stack size and depending on any future limitation on transaction size. For instance, inside `ExpressionIdentifier`, the `inner_relabel` function, which relabels the parsed expression tree is recursive and does not limit recursion depth. In case of `ExpressionIdentifier::run_pass`, less than `10000` characters in contract code were sufficient to cause a stack overflow.

It should be noted that the `MAX_CALL_STACK_DEPTH` check does not mitigate this issue, as the issue happens during the static analysis phase.

Reproduction Steps Generate the AST tree for the `(((...)))` expression and process it:

`python -c "print('(' * 5000);" > xz.clar`

`python -c "print(')' * 5000);" >> xz.clar`

`clarity-cli check xz.clar`

Recommendation Introduce recursion depth bounds inside libraries above, and return an error if the pre-set recursion depth is surpassed.

Finding	**Denial of Service via ClarityVM Process Thrashing**
Risk	**Medium** Impact: Medium, Exploitability: Medium
Identifier	NCC-BLSK002-005
Status	New
Category	Denial of Service
Component	Clarity VM
Location	Execution environment memory handling
Impact	If the network accepts a specially crafted contract, a call to the contract's exposed function would get the node to start thrashing and eventually crash.

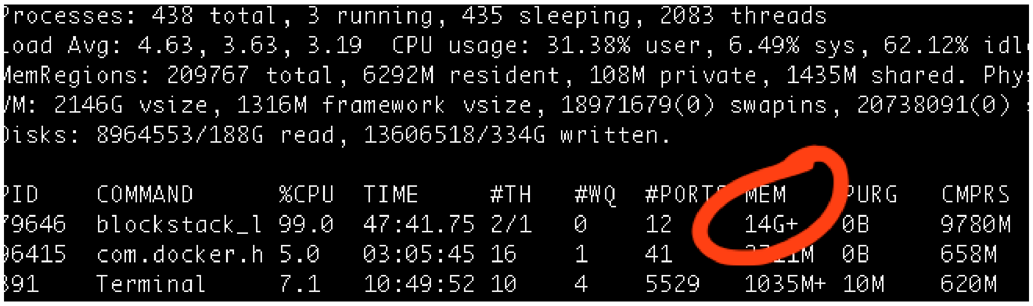
Description A maximum limit on the size of a list or buffer in Clarity is currently 1MB. A small smart contract can fill a 1MB buffer by duplicating an initial variable value. Next, the variable holding 1MB of data can be copied n times. For instance:

```
         (define-data-var XZ (buff 1048576 ) "a")
         →
          (var-set XZ (concat (var-get XZ) (var-get XZ) ))
          →
          (var-set XZ (concat (var-get XZ) (var-get XZ) ))
          →
          (var-set XZ (concat (var-get XZ) (var-get XZ) ))
          →
          (var-set XZ (concat (var-get XZ) (var-get XZ) ))
          →
          [...]
          (var-set XZ (concat (var-get XZ) (var-get XZ) ))
          →
          (var-set XZ (concat (var-get XZ) (var-get XZ) ))
          →

          (print (len (var-get XZ)))       ; prints 1024*1024 = 1048576

          (define-data-var var1 (buff 1048576) (var-get XZ))
          →
          (define-data-var var2 (buff 1048576) (var-get XZ))
          →
          [...]
          (define-data-var varN (buff 1048576) (var-get XZ))
```

While this issue may be remediated by enforcing a limit on transaction size (the PoC contract was less than 50 kilobytes) or by estimating the transaction's cost before execution, it would be prudent to add a sanity check on the amount of memory the runtime can allocate.



```
rocesses: 438 total, 3 running, 435 sleeping, 2083 threads
oad Avg: 4.63, 3.63, 3.19  CPU usage: 31.38% user, 6.49% sys, 62.12% idl
emRegions: 209767 total, 6292M resident, 108M private, 1435M shared. Phy
M: 2146G vsize, 1316M framework vsize, 18971679(0) swapins, 20738091(0)
isks: 8964553/188G read, 13606518/334G written.

ID     COMMAND      %CPU  TIME     #TH    #WQ   #PORT  MEM     PURG   CMPRS
9646   blockstack_l 99.0  47:41.75 2/1    0     12     14G+    0B     9780M
6415   com.docker.h 5.0   03:05:45 16     1     41     [illegible]1M  0B     658M
91     Terminal     7.1   10:49:52 10     4     5529   1035M+  10M    620M
```

Recommendation In addition to a particular limiting the size of memory values, enforce a limit on the overall memory that a runtime can hold.

Finding	**Discrepancies Between SIP 005 and Implementation**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-BLSK002-002
Status	New
Category	Other
Component	Transaction validation
Location	https://github.com/blockstack/blockstack-core/blob/222f545b/src/chainstate/stacks/mod.rs#L201 https://github.com/blockstack/blockstack-core/blob/222f545b/src/chainstate/stacks/auth.rs#L249
Impact	If a reimplementation of the Stacks blockchain client follows SIP 005, it risks being forked off the main network.

Description

In SIP 005, in the Transaction Authorization section, the authorization type value is defined to be 0x03 and 0x04 for standard and sponsored transactions, respectively. In the implementation, the values 0x04 and 0x05 are in fact used:

```
pub enum TransactionAuthFlags {
    // types of auth
    AuthStandard = 0x04,
    AuthSponsored = 0x05,
}
```

Another source of possible confusion is in the context of multi-sig transactions. When validating a multi-sig transaction, the number of *signatures* inside the mentioned vector is limited to 65535 (note the usage of `u16::checked_add` in the snippet below), which could be considered an anti-DoS measure. However, the number of public keys in the same vector is unlimited and is only bounded by any higher-level transaction size eventual limit. See the `verify` function that applies to multi-sig transactions and observe the usage of `u16::checked_add(1)` only in the case a signature is processed:

```
for field in self.fields.iter() {
    let pubkey = match field {
        TransactionAuthField::PublicKey(ref pubkey) => {
            if !pubkey.compressed() {
                have_uncompressed = true;
            }
            pubkey.clone()
        },
        TransactionAuthField::Signature(ref pubkey_encoding, ref sigbuf) => {
            if *pubkey_encoding == TransactionPublicKeyEncoding::Uncompressed {
                have_uncompressed = true;
            }

            let (pubkey, next_sighash) =
            ➜  TransactionSpendingCondition::next_verification(&cur_sighash,
            ➜  cond_code, self.fee_rate, self.nonce, pubkey_encoding, sigbuf)?;
            cur_sighash = next_sighash;
            num_sigs = num_sigs.checked_add(1).ok_or(net_error::VerifyingError(
             ➜  "Too many signatures".to_string()))?;
            pubkey
        }
    };
    pubkeys.push(pubkey);
}
```

While the limit on the number of signatures should be definitely mentioned in SIP 005, it is unclear whether the original consensus rule was intended to leave the number of public keys unbounded. Either way, it would make sense to discuss this rule as well in SIP 005.

Finally, as discussed with the Blockstack team, the rolling hash multi-sig transaction validation did not yet make it into SIP 005.

Recommendation Correct either the documentation or the implementation when it comes to authorization type values. As for the limit on the multisig length (in terms of the number of both public keys and signatures) discuss this constraint in SIP 005.

Finding	**Reward Address Duplication and SIP-007 Inaccuracy**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-BLSK002-007
Status	Reported
Category	Cryptography
Component	SIP-007
Location	`LeaderKeyRegisterOp::parse_data`
Impact	When it comes to PoX reward address computation, SIP-007 diverges from the implementation, resulting in misinformation on how rewards work and complicating a possible re-implementation.

Description According to the “Leader Block Commits” section of SIP-007:

> a. PoX recipients are chosen as described in “Stacking Consensus Algorithm”: addresses are chosen without replacement, by using the previous burn block’s sortition hash, mixed with the previous burn block’s burn header hash as the seed for the ChaCha12 pseudorandom function to select M addresses. b. The leader block commit transaction must use the selected M addresses as outputs [1, M] That is, the second through (M+1)th output correspond to the select PoX addresses. The order of these addresses does not matter. Each of these outputs must receive the same amount of BTC. c. If the number of remaining addresses in the reward set N is less than M, then the leader block commit transaction must burn BTC: i. If N > 0, then the (N+2)nd output must be a burn output, and it must burn (M-N) * (the amount of BTC transfered to each of the first N outputs) ii. If N == 0, then the 2nd output must be a burn output, and the amount burned by this output will be counted as the amount committed to by the block commit

The implementation fixes `M` to 1 and does away with what’s described in the paragraph above. In particular, the PoX address is not necessarily the second address in the transaction and the cases where the reward set N is less than M does not apply. In addition, replacement was not identified in the code, as such, at the moment of writing, it appears that addresses are replaced and can be duplicated over blocks in the same reward cycle.

```
        for (ix, output) in outputs.into_iter().enumerate() {
            // only look at the first OUTPUTS_PER_COMMIT outputs
            //   or until first _burn_ output
            if ix >= OUTPUTS_PER_COMMIT {
                break;
            }
            if output.address.is_burn() {
                burn_fee.replace(output.amount);
                break;
            } else {
                // all pox outputs must have the same fee
                if let Some(pox_fee) = pox_fee {
                    if output.amount != pox_fee {
                        warn!("Invalid commit tx: different output amounts for di
                         ➜  fferent PoX reward addresses");
                        return Err(op_error::ParseError);
                    }
                } else {
                    pox_fee.replace(output.amount);
                }
                commit_outs.push(output.address);    }}
```

Recommendation Redo the Leader Block Commits section in the SIP to correspond exactly to what's implemented by the code (M=1, the PoX transaction is not necessarily the second one, remove unimplemented discussion around N<M etc).

Finding	**Block and Transaction Encoding Tolerates Arbitrary Suffix**
Risk	**Informational** Impact: Undetermined, Exploitability: Undetermined
Identifier	NCC-BLSK002-003
Status	New
Category	Other
Location	`net/codec.rs`
Impact	If messages are not deserialized and then serialized again before being stored or forwarded to other nodes, DoS concerns may arise.

Description Stacks 2.0 uses a custom serialization scheme, akin to Bitcoin's object serialization/deserialization method. For instance, a `StacksTransaction` deserialization is as follows:

```
fn deserialize(buf: &Vec<u8>, index_ptr: &mut u32, max_size: u32) ->
➜  Result<StacksTransaction, net_error> {
    let mut index = *index_ptr;

    let version_u8 : u8              = read_next(buf, &mut index, max_size)?;
    let chain_id : u32               = read_next(buf, &mut index, max_size)?;
    let auth : TransactionAuth       = read_next(buf, &mut index, max_size)?;
    let anchor_mode_u8 : u8          = read_next(buf, &mut index, max_size)?;
    let post_condition_mode_u8 : u8 = read_next(buf, &mut index, max_size)?;
    let post_conditions : Vec<TransactionPostCondition> = read_next(buf, &mut
    ➜  index, max_size)?;
    let payload : TransactionPayload = read_next(buf, &mut index, max_size)?;
   [...]
```

If the `buf` variable contains data after `TransactionPayload`, this data will be ignored. If the future P2P layer (currently out of scope) does not return an error on existence of suffix data, nodes will tolerate blockchain messages with junk suffixes. This was a cause of CVE-2013-4627, since Bitcoin also had an optimization which saved and forwarded serialized data as opposed to deserializing and then serializing them before storage/validation.

More context on the Bitcoin issue: in 2013 the Bitcoin network was attacked using a related vector[3] which became CVE-2013-4627.[4] At the time, the Bitcoin nodes would broadcast messages as they were received from the network, without re-serializing them first. Due to Bitcoin's custom transaction serialization scheme, a transaction message could contain data *after* the serialized transaction, which the Bitcoin client would ignore, but at the same time save the full message and broadcast it in that form. See also this StackExchange question.[5]

Recommendation Blockstack may consider rejecting network messages with suffix after parsed serialized objects and this may be handled on the P2P network processing layer.

[3] https://bitcointalk.org/index.php?topic=259101.msg2763875#msg2763875
[4] https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures#CVE-2013-4627
[5] Serialized transaction bigger than the actual transaction object? https://bitcoin.stackexchange.com/questions/83485/serialized-transaction-bigger-than-the-actual-transaction-object-cve-2013-4627

Finding: SEED_NORM Calculation Does Not Follow SIP 001

Risk **Informational** Impact: Low, Exploitability: None

Identifier NCC-BLSK002-006

Status Reported

Category Cryptography

Component SIP 001

Location `SortitionHash::mix_burn_header`

`SortitionHash::mix_VRF_seed`

Impact SIP 001 readers will be misinformed on how the `SEED_NORM` quantity (used to pick the sortition winner) is computed.

Description In the pseudo-code listed in SIP 001, the `select_block` function computes the `SEED_NORM` as follows:

```
SEED_NORM = num(hash(SEED || BURN_BLOCK_HEADER.nonce)) / TOTAL_BURNS
```

The implementation, on the other hand, computes it roughly as follows:

```
sha256( sha256(prev_sortition_hash || burn header hash) || VRF seed )
```

In particular, the previous sortition hash is first mixed with the burn block header hash. Next, the resulting blob is mixed with the VRF seed, before being converted to a 256-bit representation. Finally, the interval corresponding to the obtain number is what determines the winner:

```
        let index = sortition_hash.mix_VRF_seed(VRF_seed).to_uint256();
         for i in 0..dist.len() {
             if (dist[i].range_start <= index) && (index < dist[i].range_end) {
                 debug!(
                     "Sampled {}: sortition index = {}",
                     dist[i].candidate.block_header_hash, &index
                 );
                 return Some(i);
             }
         }
```

Recommendation If SIP-001 cannot be corrected at this point, consider publishing an erratum on SIPs and mention this discrepancy.

Appendix A: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

Appendix B: Project Contacts

The team from NCC Group has the following primary members:

- Aleksandar Kircanski — Consultant
  aleksandar.kircanski@nccgroup.com
- Gérald Doussot — Consultant
  gerald.doussot@nccgroup.com
- Javed Samuel — Cryptography Services Director
  javed.samuel@nccgroup.com

The team from Blockstack has the following primary members:

- Diwaker Gupta — Blockstack
  diwaker@blockstack.com
- Jude Nelson — Blockstack
  jude@blockstack.com
- Ludovic Galabru — Blockstack
  ludovic@blockstack.com

Appendix C: SIP-001 and SIP-007 Testing Methodology

Over the 5-day course of the review, NCC consultant's primary testing strategy was to match the code against the SIPs, to rely on unit test modification (where needed) and to look for:

Intended behavior vs. actually implemented behavior: If the (assumed) implementation's intent should behave is different than what the implementation actually does, this would clearly be an issue. Typically this happens in edge-cases, either in user-controllable input or, more interestingly, in *edge-case state*. For example, `handle_new_burnchain_block` processing may need to catch up with a number of burnchain blocks and the to-be-processed burn chain blocks may be describing a degenerate sequence of Stacks blocks (e.g. half-valid Stacks block information, high number of Stacks forks, etc).

Usage of panicking methods: Panicking checks such as `expect` and `unwrap` are sprinkled throughout the code to test for invariants whose negation would signal a corrupt state. NCC Group looked for an instance where a non-invariant slipped through and is checked by a panicking statement. For instance, `tuple_to_pox_addr` will panic if the tuple (previously retrieved from the contract) has an invalid version, however, the contract itself contains an address version check.

Integer underflow and overflow: Unsafe arithmetic, e.g., when user burn amounts inside a burn chain block are accumulated, plain addition is used (this is a non-issue due to u128 size vs. Satoshi amount accumulation).

Observations

The sortition procedure includes creating a `distribution` (interval decomposition based on user burns) and identifying the winning sub-interval. In SIP-001, the sub-interval is computed using the `SEED_NORM` pseudo-random value. As discussed in [finding:112e9b], the `SEED_NORM` is computed differently than what's described by SIP-001. An erratum for SIP 001 (which can be included in one of the next SIPs) can be published to correct this issue.

Computing the distribution relies on fixed-point arithmetic. This is not mentioned in SIP-001 and may be mentioned in the erratum for completeness purposes.

While user burns are discussed in the Burning Pools section of SIP-001, the content of the user burn transactions is not mentioned in the Election Protocol section, even though user burns are an integral part of the election.

Reward address set computation is unique in the sense that even though it is implemented on the core blockchain layer, it calls out to the layer above itself (the smart contract layer). It does so by using a fixed boot smart contract address and by concatenating Clarity statements, such as:

```
    fn get_balance(peer: &mut TestPeer, addr: &PrincipalData) -> u128 {
        let value = eval_at_tip(
            peer,
            "pox",
            &format!("(stx-get-balance '{})", addr.to_string()),
        );
```

If `addr.to_string()` could contain arbitrary characters, Clarity statement injection would be possible. However, since `PrincipalData` serialization cannot contain special Clarity characters, this is non-exploitable. If Clarity contracts' exposed functions can be referenced at the core level (e.g., via a dynamic list of active contracts), such an approach would roughly be similar to SQL prepared statements as opposed to SQL statement concatenation and would mitigate any injection potential in future revisions of the code.

As discussed with the Blockstack team, reward address sampling has been refactored and this resulted in the implementation diverging from SIP-007, as described in [finding:b047e8]. Of note is that SIP-007 explains that addresses are sampled with replacement, whereas replacement code was not identified in `blockstack_lib`, which opens up the question of whether the final implementation should perform sampling with replacement or without replacement.

Descriptive observations

In the distribution and sortition calculation, two relevant methods are `make_distribution` which transforms burns into an interval decomposition and `select_winning_block` which uses the interval decomposition to perform a random weighted sampling of the winning block.

As for anchor block identification, to ensure that a large fraction of Stacks mining power received the anchor block, SIP 007 introduces a method for determining the anchor block. On the SIP-007 side, given a recent window of `w` Bitcoin blocks, SIP 007's `find_anchor_block` inspects a sub-graph of the Stacks chain DAG. On the one hand, in a non-degenerate case, the sub-graph pertaining to the `PREPARE` phase may be a single chain. On the other hand, the `PREPARE` sub-graph could be set of independent DAGs (connected by Stacks blocks outside the `w`-window picture). If one walks from all the leaf `PREPARE` sub-graph nodes towards the root of the chain, `find_anchor_block` identifies the most visited block, assuming this number is strictly greater than `w/2`. In such a case, there can only exist one such node. This node is the anchor block.

Up to vagueness inherent to pseudo-code, the `get_chosen_pox_anchor` function closely follows the SIP-007 specification. There is one optimization in the code that is not mentioned in SIP-007: finding nodes' ancestors outside the window (the `last_ancestor_of_block_before` function) implemented statefully in the code. During the node walk, if a Bitcoin block that has already been processed is encountered, the ancestor is pulled from the `memoized_candidates` hash map. This optimization makes processing linear complexity and does not appear to introduce any unintended behavior.

Appendix D: Hiding Map Value from Proof Unit Test Case

The test case below illustrates how one can abuse the proof verification method to hide the current map value, as described in finding NCC-BLSK002-001 on page 6. Add this test case at the end of the proof.rs file and run with the `cargo test chainstate::stacks::index::proofs::test::ncc_verifier_catches_stale_proof -- --nocapture` command in a terminal shell.

The proof verification wrongly returns that it validated value `old_v` is in block 5 root hash, while it is actually set to `another_v`.

```
#[test]
    fn ncc_verifier_catches_stale_proof() {
       // use std::env;
       // env::set_var("BLOCKSTACK_TEST_PROOF_ALLOW_INVALID", "1");

        let path = "/tmp/ncc_rust_marf_verifier_catches_stale_proof".to_string();
        match fs::metadata(&path) {
            Ok(_) => {
                fs::remove_dir_all(&path).unwrap();
            },
            Err(_) => {}
        };

        let mut m = MARF::from_path(&path, None).unwrap();

        let sentinel_block = TrieFileStorage::block_sentinel();
        let block_0 = BlockHeaderHash([0u8; 32]);
        let block_1 = BlockHeaderHash([1u8; 32]);
        let block_2 = BlockHeaderHash([2u8; 32]);
        let block_3 = BlockHeaderHash([3u8; 32]);
        let block_4 = BlockHeaderHash([4u8; 32]);
        let block_5 = BlockHeaderHash([5u8; 32]);


        let k1 = "K1".to_string();
        let old_v = "OLD".to_string();
        let new_v = "NEW".to_string();
        let new_new_v = "NEWNEW".to_string();
        let new_new_new_v = "NEWNEWNEW".to_string();
        let another_v = "ANOTHERV".to_string();

        m.begin(&sentinel_block, &block_0).unwrap();
        m.commit().unwrap();

        // Block #1
        m.begin(&block_0, &block_1).unwrap();
        let r = m.insert(&k1, MARFValue::from_value(&new_v));
        let (_, root_hash_1) = Trie::read_root(m.borrow_storage_backend()).unwrap();
        m.commit().unwrap();

        // Block #2
        m.begin(&block_1, &block_2).unwrap();
        let r = m.insert(&k1, MARFValue::from_value(&old_v));
        let (_, root_hash_2) = Trie::read_root(m.borrow_storage_backend()).unwrap();
        m.commit().unwrap();

        // Block #3
        m.begin(&block_2, &block_3).unwrap();
        let r = m.insert(&k1, MARFValue::from_value(&new_new_v));
        let (_, root_hash_3) = Trie::read_root(m.borrow_storage_backend()).unwrap();
        m.commit().unwrap();

        // Block #4
        m.begin(&block_3, &block_4).unwrap();
        let r = m.insert(&k1, MARFValue::from_value(&new_v));
        let (_, root_hash_4) = Trie::read_root(m.borrow_storage_backend()).unwrap();
        m.commit().unwrap();

         // Block #5
         m.begin(&block_4, &block_5).unwrap();
         let r = m.insert(&k1, MARFValue::from_value(&another_v));
         let (_, root_hash_5) = Trie::read_root(m.borrow_storage_backend()).unwrap();
         m.commit().unwrap();

        merkle_test_marf_key_value(m.borrow_storage_backend(), &block_5, &k1, &another_v, None);
        merkle_test_marf_key_value(m.borrow_storage_backend(), &block_2, &k1, &old_v, None);

        let root_to_block = m.borrow_storage_backend().read_root_to_block_table().unwrap();

        // prepare a proof for the wrong root hash i.e. block2 instead of block5
        println!("DEBUG: building proof5()");
        let proof_5 = TrieMerkleProof::from_entry(m.borrow_storage_backend(), &k1, &old_v,
         ➜  &block_2).unwrap();

        //println!("DEBUG proof_5: {:?}", proof_5);

        // Send proof to victim
        // Verification succeeds when it should not
        let triepath_4 = TriePath::from_key(&k1);
        let marf_value_4 = MARFValue::from_value(&old_v);
        let block_map = m.borrow_storage_backend().block_map.clone();
        println!("DEBUG: verify()");
        assert!(proof_5.verify(&triepath_4, &marf_value_4, &root_hash_5, &root_to_block));

    }
```