Monero multi-signature patch review

rino

inference

20220627 – FINAL

Contents

1 Summary

RINO solicited Inference to review changes to the multisignature system of the Monero cryptocurrency. These changes were combined into a pull request which sought mainly to address 3 vulnerabilities, two of which had been reported anonymously. We reviewed the changes in the patch with a particular focus on assessing how these vulnerabilities were addressed.

This report presents the work performed, and notably describes:

- The 3 vulnerabilities the patch sought to fix—plus one it did not—along with an assessment of the extent to which the issues have been addressed.
- 7 potential security issues,
- 4 observations related to defense-in-depth, reliability, and performance.

We would like to thank RINO for trusting us.

2 Project overview

2.1 Scope

The client requested a review of this patch to the multisig system, in pull request #8149. The code was reviewed at commit f5e33479d656bc95001d2f135651e9fe9194681a.

We reviewed the changes introduced by this patch, which affected the following files:

- `src/cryptonote_config.h`
- `src/cryptonote_core/cryptonote_tx_utils.{h, cpp}`
- `src/multisig/multisig_clsag_context.{h, cpp}`
- `src/multisig/multisig_tx_builder_ringct.{h, cpp}`
- `src/ringct/rct.{h, cpp}`
- `src/wallet/wallet2.{h, cpp}`

We also assessed the three vulnerabilities the patch sought to address, looking at their security impact, and the extent to which they have been adequately addressed by the changes in the patch.

2.2 Goals

Patch #8149 attempts to fix three vulnerabilities related to Monero's multisig system. Two of these were reported anonymously through a bug bounty platform, in the form of a patch providing proof of concept attacks.

The threat model we use is the standard one, where the attacker controls a subset of the parties holding a collective multisignature wallet. The attacker controls an insufficient number of parties in order to sign transactions without the consent of other participants. The main security goals in this model are to prevent unauthorized transactions from being signed, while also allowing honest participants to continue signing transactions.

3 Vulnerability assessment

3.1 Drijvers attack

This attack was a general attack on two-round multisignature schemes based on Schnorr signatures, from a 2018 paper by Drijvers et al. This includes the CLSAG scheme that Monero uses. The attack allows an attacker to forge a signature on a message of their choice, by initiating multiple signing sessions in parallel.

The idea is that in a simple multisignature scheme, each participant computes a random $\alpha_i \in_R \mathbb{Z}_q$, and then broadcasts $A_i := \alpha_i \cdot G$ to the other participants, where G is a group generator. The participants then compute $A := \sum_i A_i$, to create a commitment to their joint nonce $\sum_i \alpha_i$.

The issue stems from the fact that by going last, a malicious participant can choose their nonce α_i as a function of the commitments A_j that it has already seen, and thus gain a significant influence over the final result.

The general process for attacks based on Drijvers is as follows:

1. The attacker waits until the other participants have sent their A_i^j commitments, for each parallel signing session.
2. The attacker defines their α_k^j values based on the commitments from the other participants, across all sessions, and on the message they're computing a forgery over.
3. The attacker waits to receive all of the signatures in each session, and then uses them to forge a signature over their message.

Crucially, step 2 depends on the fact that the A_i^j commitments are fixed even if α_k^j changes.

3.1.1 Impact

The result of this attack is that if enough concurrent signing sessions are initiated, it's possible to forge a signature for a different message.

Using the advanced attack from On the (in)security of ROS, a signature could be efficiently forged using 256 concurrent multisig sessions. Using fewer sessions would still allow forgery, although would

require significantly more effort. Using 256 sessions works perfectly because scalars for Curve25519 can be represented with 256 bits.

It also seems that the attack could plausibly be deployed. Because of the asynchronous nature of the multisig system, it's possible for several multisig attempts on different messages to exist at the same time. If an attacker controlled two participants in the consortium, they could initiate many legitimate transactions with one participant, and then use the other to carry out the attack, forging a signature on a message of their choice without the consent of the other participants.

Because of these factors, the impact of this attack is high.

3.1.2 Patch

This patch sought to address this issue by applying the fix used in MuSig2, a multisignature scheme for Schnorr signatures.

The idea is that instead of each participant having a single nonce α_i, the participants have several nonces $\alpha_i^1, \ldots, \alpha_i^m$, along with corresponding commitments $A_i^j := \alpha_i^j \cdot G$. Rather than simply calculating $A := \sum_{i,j} A_i^j$, the participants first calculate an accumulation factor $b := H(\{A_i^j\})$, by hashing all of the commitments, and then calculate:

$$A := \sum_i \sum_j b^j A_i^j$$

This approach effectively mitigates the Drijvers attack, by making the contribution of each participant depend on the contribution of all participants, via the factor b.

3.1.3 Assessment

We checked that the patch implemented this approach correctly. In effect, the patch uses two nonces for each participant, which is sufficient. (We note that the patch technically allows a dynamic number of nonces for each participants in many parts of the code, but then hardcodes the number to 2). The patch also uses domain separation for the hash calculating b, and also includes as much contextual information about the transaction as possible, mitigating potential attacks similar to the Frozen Heart class of vulnerabilities, which rely on insufficient information being included in the hash. Not including this information in the hash might also enable Drijvers attacks which would work by varying the contextual information rather than the nonces.

Because of this, we assess this vulnerability to have been adequately addressed.

3.2 Nonce reuse

This was the first of two issues reported anonymously.

The multisig system in Monero has an initial phase in which nonces $k \in_R \mathbb{F}_q$ are generated, and then commitments $K := k \cdot G$ are sent to other participants. This phase can be performed multiple times, preparing nonces for many signatures in advance. When a message arrives that a participant would like to sign, they create a transaction, which includes information instructing other participants on which nonces to use: by including a nonce commitment K instead the transaction, they indicate that the corresponding nonce k needs to be used. A participant can then use this public K value to look up the corresponding k scalar, using a map of nonces that they had prepared before.

The issue stemmed from the fact that the transaction wasn't validated to not instruct participants to use a nonce multiple times, and so a malicious transaction prepare could effectively make other participants reuse nonces, and thus recover private keys.

3.2.1 Impact

Applications making use of the multisig system need to do extra validation to make sure that the transaction is something they want to sign. It's possible that this validation would have also checked that these nonce commitments weren't duplicated, but this seems unlikely.

If successfully exploited, this vulnerabity would allow the recovery of private keys corresponding to transaction outputs, and thus unauthorized spending of funds.

Because of these factors, the impact of this attack is high.

3.2.2 Patch

The patch works by always erasing the memory inside of this map of nonces anytime a nonce is retrieved, replacing that nonce with 0. Furthermore, nonces returned are checked to not be 0, and an error is returned instead, to prevent continuing the signing process with a 0 nonce.

3.2.3 Assessment

This change prevents any duplication from causing nonce reuse, by preventing reuse at the lowest level of the nonce system, rather than validating the transaction for reuse.

We assess this vulnerability to have been adequately addressed.

3.3 Insufficient transaction validation

This was the second of two issues reported anonymously.

The multisig system works in several phases as mentioned previously. Using the notation in Zero to Monero, Section 3.6, in the final phase, the initiator of the transaction has aggregated the nonce commitments, and calculated the values $c_1, \ldots, c_n$ and $r_1, \ldots, r_n$, with the exception of r_π. All that's left is to calculate:

$$r_\pi := \sum_i \alpha_i - c_\pi w_{\pi,i}$$

using the contributions from each participant. This is done in a round-robin fashion, with each participant contributing an additional term in this sum, and then forwarding the transaction further.

The issue was that the c_π value wasn't validated at all, and participants would blindly calculate $\alpha_i - c_\pi w_{\pi,i}$ without any regard as to whether or not this was actually related to the transaction presented to them.

3.3.1 Impact

The impact of this issue is complicated, and difficult to disentangle from the fact that transactions in general may not be sufficiently validated at the application layer.

In particular, you could choose c_π originating from a different transaction, and thus get the participants to authorize an action different from the one described and validated at the application level.

Because of these factors, the impact of this attack is high.

3.3.2 Patch

The patch works by completely rewriting the transaction building system.

The goal of this rewrite is to recompute more components of the transaction from the application level details, like which outputs are being spent. In effect, the new system tries to independently recompute the details of the transaction, rather than trusting the values inside of the transaction as presented.

3.3.3 Assessment

We audited this new system, and had several findings and observations presented further in this report.

In regards to this particular vulnerability, we assess it to have been addressed by this patch.

While the direct issue of smuggling an unrelated c_π value has been patched, the larger issue of insufficient application validation of transactions still remains. Looking at the RPC functionality, it's of utmost importance that applications validate transactions themselves, before invoking the RPCs. Applications need to validate that the transactions correspond to actions that they actually want to take, and don't have adverse side effects like spending the wrong funds.

3.4 Burning bug

Monero's 2018 burning bug is another issue brought to our attention during the audit. In essence, the issue is that the output of a transaction may be a stealth address that has been repeated, or even already spent. This has the effect of burning funds, since an output can only be spent once, so the repeated outputs are rendered unusable.

While this issue had been fixed for standard wallets, concerns were raised about whether or not it might be present in multisig wallets. Patch #8149 initially contained a proposed fix for this issue, which was subsequently removed from the commits, for unclear reasons.

The burning bug was fixed through additional checks inside of wallets, but it is only one instance of a more general class of risks. For example, a similar issue remains in the context of multisignatures, where a malicious participant can initiate a multisignature using duplicate or repeated outputs. If the other participants sign this transaction, completing the signature, then it would burn the funds held collectively.

3.4.1 Impact

The impact of the burning bug, if not mitigated at the application layer, would allow a single participant in a multisig wallet to unilaterally burn funds held collectively by the group of participants in the wallet.

The impact is thus also high.

3.4.2 Assessment

As previously mentioned, the patch no longer contains a fix for the burning bug, so the code under review does not include any more controls to prevent its exploitation.

To understand how to best mitigate such an issue, note that creating a multisignature makes use of two essential layers:

- The first, upper layer involves creating the *content of a transaction*, or agreeing to it. This content specifies what effect the transaction is supposed to have, and applications approve the transactions based on whether or not they want this effect to happen. For example, a transaction might describe sending coins owned by the multisig group to another address, and applications would have to decide if this transaction should be approved.
- The second, lower layer involves collectively *creating a signature* over this transaction, and is where the multisig system itself is used. The end result should be a valid signature for that transaction, produced with the cooperation of several participants. Crucially, this layer should not be able to produce a signature for a different transaction than the one approved by the layer above it. Values derived from the high-level values of the transaction, like the description of what coins are being spent, may be independently checked for integrity, since those high-level values are what applications have validated.

That is, the lower-level code should not necessarily trust its callers at the application level: it should not assume any pre-condition on the input values, except for those that it cannot verify.

The previous vulnerability discussed, “Insufficient transaction validation”, stemmed from the fact that the multisig system didn’t check the correctness of intermediate values derived from the high-level values in the transaction. This would mean that even if an application approved a transaction, in the first layer, the second layer, involving the multisig system, could be tampered with to change the transaction being signed.

The burning bug, on the other hand, is an issue in the first layer. The root cause is applications not validating that a transaction does not contain these malicious repeated outputs. It is possible to mitigate the issue by moving more responsibilities from the first layer to the second, and thus reducing which parts of a transaction the first layer approves. For example, some proposals modify the way outputs are created in a multisig context, in order to have multiple participants collaborate in their creation, rather than being proposed by a single participant.

Unfortunately, this kind of shifting of responsibility will never be complete; there will always be aspects of transactions which cannot be verified by the multisig system itself. For example, a transaction could describe the transfer of funds to a perfectly legitimate address. In this case, the multisig system has no way of knowing whether or not the applications making use of it want the transfer to happen. This transfer has the same appearance regardless of whether or not the participants would like it to happen.

We stress that while some vulnerabilities in the application layer could be mitigated by shifting more responsibility to the multisig system, it will always be necessary for applications to validate that the transactions they’re signing perform the right actions. Having a clear delineation of responsibilities betwen the application layer and the multisig layer also facilitates their security analysis.

Ultimately, such issues are best avoided by clearly defining and documenting the system's security model: shared responsibilites, pre- and post-conditions, redundant controls, and last but not least properly testing such controls.

4 Security issues

Severities are quantified with two dimensions, roughly defined as follows:

- Exploitability:
  - Low: Other vulnerabilities are needed to exploit the bug.
  - Medium: Privileged access is needed (for example, local), or costly attack (in terms of computation, storage, bandwidth, etc.)
  - High: Exploitation is easy (for example, remote, unauthenticated)
- Impact
  - Low: The vulnerability exploit has clearly little to no impact on the system, its users, and operators.
  - Medium: Not low, but not high.
  - High: Critical system assets are clearly impacted, in terms of confidentiality, integrity, availability, or value.

Note that these definitions are vague on purpose, as they depend on the business context and assets at stake. For example, if service availability is critical to the system, then a DoS can qualify as high-impact.

4.1 S-MSG-001: Hash-to-scalar modulo bias

Exploitability: low

Impact: low

4.1.1 Description

In `src/ringct/rctOps.cpp` the function `hash_to_scalar()` works by using a hash function, viewing the 32-byte output as an unsigned integer, and then reducing this output modulo the order of the group:

```
void hash_to_scalar(key &hash, const void * data, const std::size_t l)
    {
     cn_fast_hash(hash, data, l);
     sc_reduce32(hash.bytes);
}
```

Because the order of the Curve25519 group, close (and less than) 2^{256}, is not a power of 2, this reduction introduces a small modulo bias in the output, making hashes have a non-uniform distribution.

Exploitability: This function is used many times throughout the construction of multisig transactions, with data either controlled or influenced by other participants in the protocol.

Impact: The modulo bias is negligible, and the slightly non-uniform distribution will not affect the security of the protocol.

4.1.2 Recommendation

Adding an extra 128 bits of output to the hash would ensure that the bias is $\leq 2^{-128}$. This would bring the output of the hash to 384 bits. Alternatively, a 512 bit hash could be used, as is done in the Ed25519 signature scheme. It's possible that code to do this reduction could be reused in this context.

The post The definitive guide to "modulo bias and how to avoid it"! is a good reference to mitigate this class of issues.

4.2 S-MSG-002: Lack of domain separation in lists hashing

Exploitability: medium

Impact: low

4.2.1 Description

in `src/multisig/multisig_clsag_context.cpp`, many hash functions take in variable-length lists of elements, but don't include any information about their length, or any kind of domain separation.

For example, consider the `b_params` values, which are hashed by concatenation:

```
// musig2-style nonce combination factor components for multisig
    signing
//   b = H(domain-separator, {P}, {C}, C_offset, message, {
    L_combined_alphas}, {R_combined_alphas}, I, D, {s_non_l}, l)
//   - {P} = ring of one-time addresses
```

```
4  //   - {C} = ring of amount commitments (1:1 with one-time addresses)
5  //   - C_offset = pseudo-output commitment to offset all amount
       commitments with
6  //   - message = message the CLSAG will sign
7  //   - {L_combined_alphas} = set of summed-together public nonces from
       all multisig signers for this CLSAG's L component
8  //   - {R_combined_alphas} = set of summed-together public nonces from
       all multisig signers for this CLSAG's R component
9  //   - I = key image for one-time address at {P}[l]
10 //   - D = auxiliary key image for the offsetted amount commitment '{C
       }[l] - C_offset'
11 //   - {s_non_l} = fake responses for this proof
12 //   - l = real signing index in {P} and '{C} - C_offset'
13 rct::keyV b_params;
```

Here `P`, `C`, `L_combined_alphas`, and `R_combined_alphas` are all variable length lists. Because the list of these lists is not included inside the hash, or is there any kind of separator between the lists, it's possible to have distinct inputs create a collision, by interpreting the size of the lists differently.

This issue also applies to `c_params`, `mu_C_params` and `mu_P_params` in the same file.

Exploitability: It's easy to find example inputs which cause this issue, but validation of the transaction at the application level may already be able to filter these out. There's no clear attack vector as a result.

Impact: If colliding inputs could be found, a first set of (secure) parameters could be replaced with (potentially insecure) parameters, or the other way, leading to an insecure state.

4.2.2 Recommendation

The simplest way to avoid this issue is to include the length of each variable length list before that list, inside of the hash. It might be possible that applications already depend on the current format of the hash, such as for `c_params` which is part of the signature scheme. For `b_params`, since this is just used internally for nonce aggregation in the multisig, the hash method can safely be changed.

4.3 S-MSG-003: Uncaught exceptions in clsag_context

Exploitability: high

Impact: low

4.3.1 Description

In `src/multisig/multisig_clsag_context.cpp`, the functions in this file call out to other methods which throw exceptions on invalid input, but these exceptions aren't caught inside of the functions. This means that instead of return **`false`** on invalid inputs, sometimes these functions may throw an exception instead. Because the surrounding RPC wallet does catch exceptions, this won't cause a full crash, but could still slow down the server.

For example, the function `precomp` in `src/ringct/rctOps.cpp` is called several times:

```
1 //Does some precomputation to make addKeys3 more efficient
2 // input B a curve point and output a ge_dsmp which has precomputation
     applied
3 void precomp(ge_dsmp rv, const key & B) {
4     ge_p3 B2;
5     CHECK_AND_ASSERT_THROW_MES_L1(ge_frombytes_vartime(&B2, B.bytes) ==
          0, "ge_frombytes_vartime failed at "+boost::lexical_cast<std::
         string>(__LINE__));
6     ge_dsm_precomp(rv, &B2);
7 }
```

This is used, among other places, in `multisig_clsag_context.cpp:145`

```
1   rct::precomp(D_precomp.k, D);
```

Exploitability: Combined with `S-MSG-004`, it's possible to consistently trigger the above code path, by choosing a malicious `D` as the initial signer.

Impact: Because the wallet still checks exceptions at the RPC layer, this prevents a node from crashing. If this multisig code is used in a different context, it's possible that these exceptions may not be thought of, since the code already returns a boolean to indicate failures.

4.3.2 Recommendation

To have a consistent interface, it would be better to have the functions in `multisig_clsag_context` catch these exceptions, and return **`false`**, in order to have a single mechanism to signal failure.

4.4 S-MSG-004: Unchecked D value in transaction reconstruction

Exploitability: high

Impact: low

4.4.1 Description

In src/multisig/multisig_tx_builder_ringct.cpp, the function set_tx_rct_signatures() does not check that the D value is of the correct format, when reconstructing the signature, at line 670:

```
else {
  rv.p.CLSAGs[i].D = unsigned_tx.rct_signatures.p.CLSAGs[i].D;
  rv.p.CLSAGs[i].I = I;
  D = rct::scalarmultKey(rv.p.CLSAGs[i].D, rct::EIGHT);
}
```

This value is pulled directly from unsigned_tx, which is passed on from another participant in the multisig. No validation is done on this value.

Exploitability: Any participant can directly trigger this by modifying their D value in the transaction they pass along to other participants.

Impact: Aside from S-MSG-003, there doesn't seem to be any impact, beyond causing the multisignature to fail.

4.4.2 Recommendation

The code should verify that D was constructed as expected, by performing checks similar to the code in the not reconstruction branch.

4.5 S-MSG-005: Unchecked s value in transaction reconstruction

Exploitability: low

Impact: low

4.5.1 Description

In src/multisig/multisig_tx_builder_ringct.cpp, the function set_tx_rct_signatures() does not check the s values when reconstructing the signature, beyond verifying their length, at line 644:

```
else {
  if (ring_size != unsigned_tx.rct_signatures.p.CLSAGs[i].s.size())
    return false;
  s = unsigned_tx.rct_signatures.p.CLSAGs[i].s;
}
```

This value is pulled directly from `unsigned_tx`, which is passed on from another participant in the multisig. No validation is done on this value.

Exploitability: Any participant can directly trigger this by modifying their `s` value in the transaction they pass along.

Impact: The only impact beyond that of the protocol is equivalent to revealing the secret index in the ring signature, which a malicious participant can already do.

4.5.2 Recommendation

The code here can't check that the values were generated randomly, as expected, beyond checking for trivial values like `0`. On the other hand, the protocol could be amended so that each participant in the multisig contributes to these `s` values, so that the result is random so long as at least one signer is honest.

4.6 S-MSG-006: Integer overflow in transaction fee computation

Exploitability: low

Impact: low

4.6.1 Description

In `src/multisig/multisig_tx_builder_ringct.cpp`, the function `compute_tx_fee()` could trigger an integer overflow if enough inputs or outputs with large values are spent:

```
1  static bool compute_tx_fee(
2    const std::vector<cryptonote::tx_source_entry>& sources,
3    const std::vector<cryptonote::tx_destination_entry>& destinations,
4    std::uint64_t& fee
5  )
6  {
7    std::uint64_t in_amount = 0;
8    for (const auto& src: sources)
9      in_amount += src.amount;
10
11   std::uint64_t out_amount = 0;
12   for (const auto& dst: destinations)
13     out_amount += dst.amount;
14
15   if(out_amount > in_amount)
16     return false;
17   fee = in_amount - out_amount;
```

```
18     return true;
19   }
```

Exploitability: Provided that the transaction is verified by the application, the inputs would have to actually exist, and so would require a genuinely large amount of currency to be exploited.

Impact: The fee would be miscalculated, but wouldn't have any further impact.

4.6.2 Recommendation

Checking for overflow here in both of these sums would avoid these issues.

4.7 S-MSG-007: Integer overflow in `export_multisig()`

Exploitability: medium

Impact: low

4.7.1 Description

in `src/wallet/wallet2.cpp` the function `export_multisig()` calculates the number of transactions which need to be created using `tools::combinations_count`, which can trigger an integer overflow if enough signers are involved

At line `13440`:

```
1  size_t nlr = tools::combinations_count(m_multisig_signers.size() -
       m_multisig_threshold, m_multisig_signers.size() - 1);
```

In particular, if `m_multisig_signers.size()= 70 = 2 * m_multisig_threshold`, this will overflow the 64-bit `size_t` type.

Exploitability: The exploitability is mitigated by the fact that the number of participants in a multisig may be checked elsewhere.

Impact: The impact would be an insufficient number of transaction attempts, although the performance of the system would have already degraded significantly by the time this count has reached the maximum value for `size_t`.

4.7.2 Recommendation

Checking for overflow would be a possibility here, although other safeguards for creating large multisignature wallets might already be sufficient.

Ideally, avoiding the need to create one transaction for each possible combination of signers would allow the system to scale to larger consortiums.

5 Observations

Here we list observations and suggestions not directly about security risks, but potential improvements, "defense-in-depth", quality assurance, and performance.

5.1 O-MSG-01: Inconsistent hash-to-curve validity checks

In `src/ringct/rctOps.cpp`, the function `hash_to_p3()` attempts to convert the output of a hash function into a point on the curve. The method `ge_fromfe_frombytes_vartime()` it calls to do this doesn't fail, but instead has several `assert`s that detect invalid points. This is in contrast with other functions in `rctOps.cpp`, which instead throw exceptions or return booleans. The probability of triggering this behavior is negligeable, because the output of the hash function is effectively random. Nonetheless, it might be useful to unify the behavior of the functions in this file, for consistency.

5.2 O-MSG-02: Redundant point marshalling

In `src/multisig/multisig_clsag_context.cpp:151`, the loop involves calling `rct::precomp()` in order to unmarshall points from the compressed form, and perform some precomputation for faster scalar multiplications. The loop also involves functions like `rct::subKeys` which internally unmarshall points, before marshalling back. This means that some points are unmarshalled several time, which isn't ideal.

5.3 O-MSG-03: Single multisig threshold allowed

In `src/wallet/wallet2.cpp:9142`, multisigs with a threshold of 1 are allowed, which seems odd. These are effectively the same as sharing keys.

5.4 O-MSG-04: Concurrent multisigs impossible

As warned by the comment at `src/wallet/wallet2.cpp:9103`, concurrent multisig attempts using overlapping inputs will not work, because the same nonces may be used. This is more so a deficiency with the current round-robin protocol, rather than something involving explicit consensus from participants.

6 Disclaimer

This security assessment report (“Report”) by Inference AG (“Inference”) is solely intended for RINO (“Client”) with respect to the Report’s purpose as agreed by the Client. The Report may not be relied upon by any other party than the Client and may only be distributed to a third party or published with the Client’s consent. If the Report is published or distributed by the Client or Inference (with the Client’s approval) then it is for information purposes only and Inference does not accept or assume any responsibility or liability for any other purpose or to any other party.

Security assessments of a software or technology cannot uncover all existing vulnerabilities. Even an assessment in which no weaknesses are found is not a guarantee of a secure system. Generally, code assessments enable the discovery of vulnerabilities that were overlooked during development and show areas where additional security measures are necessary. Within the Client’s defined time frame and engagement, Inference has performed an assessment in order to discover as many vulnerabilities of the technology or software analyzed as possible. The focus of the Report’s security assessment was limited to the general items and code parts defined by the Client. The assessment shall reduce risks for the Client but in no way claims any guarantee of security or functionality of the technology or software that Inference agreed to assess. As a result, the Report does not provide any warranty or guarantee regarding the defect-free or vulnerability-free nature of the technology or software analyzed.

In addition, the Report only addresses the issues of the system and software at the time the Report was produced. The Client should be aware that blockchain technology and cryptographic assets present a high level of ongoing risk. Given the fact that inherent limitations, errors or failures in any software development process and software product exist, it is possible that even major failures or malfunctions remain undetected by the Report. Inference did not assess the underlying third party infrastructure which adds further risks. Inference relied on the correct performance and execution of the included third party technology itself.