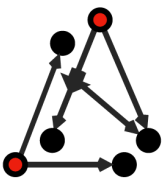
BEAM Implementation

Security Audit Report

BEAM

Report Version: 24 December 2018



Least Authority

Freedom Matters

Table of Contents

Overview

BEAM has requested that Least Authority perform a security audit of BEAM, an implementation of the MimbleWimble protocol. The audit follows the launch of Testnet, in which the code was made open source, and precedes the launch of Mainnet. The implementation is written in C++ and uses the Equihash Mining algorithm.

The audit was performed from October 1 - November 9, 2018 by Ramakrishnan Muthukrishnan, Meejah, and Jan Winkelmann. The initial report was issued on November 14, 2018. An updated report has been issued following the discussion and verification phase on December 24, 2018.

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the BEAM codebase followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code repositories are in scope:

Specifically, we examined the Git revisions:

`3a42d05f5b17731dbed042e83047f355b49e0449`

All file references in this document use Unix-style paths relative to the project's root directory.

Areas of Concern

Our investigation focused on the following areas:

- Implementation of basic cryptography (i.e. bulletproof, Pedersen, schnorr signatures etc)
- Implementation bugs (i.e. SecureErase, NoLeak, etc.)
- Logic of the node, node processor, gossip protocol and node.db file that describes SQLite structure
- Attacks that may reduce the difficulty of the writing blocks, attacks that cleverly manipulate timestamps, etc., and other consensus rules
- Significant modifications to third-party open source project code
- Anything else as identified during the initial analysis phase

Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protects users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future

audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, run time code analysis with tools like valgrind, running a few analysis tools like the gcc/clang's builtin sanitizers and other static analysis tools etc. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.

Findings

Code Quality

Overall, we found the code to be structured cleanly into different namespaces (i.e. wallet, BEAM, ECC). Additionally, there are unit tests for critical modules like node, wallet, p2p protocol, and ECC, which are an important aspect to a quality codebase.

However, we found the code somewhat difficult to review because of the coding style and suggest increasing the idiomatic use of C++. For example, in one instance, a #define-based metaprogramming pattern is used and in other cases inputs are not properly validated. While this approach may be

convenient for writing the initial version of the code, it is not convenient for reviewers reading it. We recommend that the code readability is improved as the codebase grows to facilitate easier code reviews and external contributions in the future.

Additionally, other improvements to the codebase could further reduce the risk of code errors and therefore security vulnerabilities. Such as, properly vendoring dependencies, increasing test coverage report generation and generally increasing the number of comments in the code.

Issues

We list the issues we found in the code in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Hash Processor Can Return Same Result with Different Call Sequences to Write and Operator<<	*Found to be Not Applicable*
Issue B: Write to Uninitialized Hash Function	*Partially Resolved*
Issue C: Network Encryption Leaks Abstraction Layers	*Unresolved*
Issue D: assert() Used for Input Validation	*Unresolved*
Issue E: Improper Vendoring of Dependencies	*Partially Resolved*
Suggestion 1: Use Key-Derivation Systematically	*Resolved*
Suggestion 2: Type of Loop Variables	*Partially Resolved*
Suggestion 3: Initialization of Variables	*Partially Resolved*
Suggestion 4: Use a Well-Studied Key Exchange Protocol	*Unresolved*
Suggestion 5: Do Not Implement the Key Exchange Protocol Yourself	*Unresolved*
Suggestion 6: Limit the Use of Pointer Arithmetic	*Resolved*
Suggestion 7: Use Division and Multiplication for Converting Bit/Byte Counts	*Unresolved*
Suggestion 8: Custom Curve Crypto Should Only be Used when Really Needed	*Verified: Unchanged*

Issue A: Hash Processor Can Return Same Result with Different Call Sequences to `Write` and `Operator<<`

Synopsis

Hash::Processor.Write(const char*) writes null-terminated strings, which means if a buffer is passed that contains two strings, it will return the same hash as if both strings were written consecutively. Sometimes the attacker has control over the buffer's contents, so it is able to create unexpected hashes.

Impact

It is possible for an attacker to calculate the same hash for different sequences of `Write` or `operator<<` calls.

If an attacker needs to calculate a hash they do not have all the input to (due to some of it being a secret and only known to a different node N), they may be able to trick N into creating the hash for them.

Preconditions

There needs to be a different instance of the hash processor that was fed with similar values.

Feasibility

Hash processors are used in a variety of contexts, both directly and indirectly (e.g. through the `Oracle` type). When used through an oracle, it is especially the case that formats are often similar.

Technical Details

This encoding makes no difference between `(a||b)||c` and `a||(b||c)`, which may allow attackers to create hashes that look like they were created by one hash processor but actually were created by a different one, opening doors for subtle confused deputy attacks.

Mitigation

Check that the input buffer does not contain a NULL element before the end of the string.

Remediation

Instead of using null-terminated strings, length-prefixed strings should be used.

Status

The BEAM team reports that this is, in fact, not possible. We verified their claims and reached the same conclusion. As a result, this is not longer considered a security vulnerability.

Verification

Found to be Not Applicable.

Issue B: Write to Uninitialized Hash Function

Synopsis

Reading the hash from the hash processor clears the internal state of the hash. Afterwards it needs to be either discarded or reinitialized before continuing writing to it. Instead, it is written to without reinitialization.

Impact

Use of unspecified hash function (the SHA256 function with uninitialized state).

Preconditions

Writing to a hash processor after reading from it, without re-initialization.

Feasibility

Guaranteed to happen.

Technical Details

The `>> operator` of the hash processor calls the finalize function. This zeroes the internal state of the hash function. When writing to the hash function later on, the internal state still is zeroed and the function is not a valid SHA256. None of its guarantees (wrt. preimage and collision resistance) necessarily apply.

Mitigation

Instantiate a new hash processor after reading from it and feed it with the data read from the last one. This behaves different (i.e. returns different values) to the proposed remediation, so the decision whether to mitigate or remediate is difficult to reconsider.

Remediation

Do not finalize the hash state when reading from the hash processor. Instead, make a copy of the hash state and finalize that.

Status

The BEAM team provides means to reinitialize the state of the hash function before reusing it. Using asserts, they try to make sure that they do not operate on uninitialized hash functions.

However, we advise not to use asserts to check that the state of the hash function is initialized. Asserts only have an effect during debug builds, so if this happens in a release build, it will silently return the wrong hash. We recommend either returning an error value or raising an exception.

Verification

Partially Resolved.

Issue C: Network Encryption Leaks Abstraction Layers

Synopsis

Establishing and maintaining a secure channel should be strictly separated from the application-layer protocol code. The most popular example for such a system is TLS, because it gives you the interface of a network socket and data sent and received here will be encrypted.

The way it is implemented in BEAM is that the messages for establishing the secure connection and the higher-level protocol logic are defined in the same places. The code that parses key exchange messages is the same as the one that parses Node-to-Node RPC messages.

The current system takes precautions such that before the secure connection is established, only key exchange messages are accepted, and after it is established, key exchange messages are ignored. However, a clear, architectural separation would provide much more confidence that data can not be leaked.

Even if one chooses (for any number of reasons) not to use TLS, the abstractions it gives are highly desirable for security reasons.

Impact

The current key exchange and channel encryption code is not as robust as industry standard requires it to be.

Preconditions

Does not apply.

Feasibility

Not immediately exploitable.

Technical Details

Transport encryption should behave (programmatically) like the transport it sits on top of. In this case, it is like a stream socket.

Mitigation

Mitigations have already been implemented by checking that only key exchange messages are accepted before establishing a secure channel, rather than afterwards.

Remediation

Provide a clean socket-like abstraction.

Status

It is our assessment that the BEAM team did not make significant enough changes to alleviate the risks and vulnerabilities stated above. BEAM states the following as it pertain to Issue C: *"Fixed (practically). There is a brief restriction that the only messages that can be received as a plaintext are those needed for the secure channel setup. The underlying communication object (TcpStream) is encapsulated and can not be used as-is."******. See Remarks on Abstraction, `assert` and Security below.

*****(Refer to email: "*Comments regarding the Security audit report by Least Authority, 2018-11-14*" sent on 15 December 2018)

Verification

Unresolved.

Issue D: `assert()` Used for Input Validation

Synopsis

In some instances, values are checked for unexpected values, but only in calls to `assert()`. This means that the validation does not happen in production ("Release") builds, allowing malformed data to flow through the system. Even in development ("Debug") builds, this is not an optimal solution because the program simply crashes.

A non-exhaustive list of instances of this:

- core/merkle.cpp - `Mmr::get_Proof()` - parameter `i` is only checked using assert
- core/ecc.cpp - `MultiMac::FastAux::Schedule` - parameter `nMaxOdd` is only checked using assert

- core/block_crypt.cpp - many parameters (in `IsValid`, `Pack`, `Adjust`, ..) are checked using assert; it's not clear that all callers will maintain these invariants (and there's no documentation or comments to indicate they should). Maintaining runtime checks here would be trivial.

Impact

If inputs are not properly validated and functions receive data in unexpected format, the integrity of the node state is not guaranteed.

Preconditions

The attacker provides a malicious input to our system, using a code path where validation is only performed in calls to `assert()`. Our system has to be a release build.

Feasibility

Medium.

Technical Details

`cmake` sets the NDEBUG define for release builds, which disables `assert()` calls. If these are the only validation happening, no validation is performed for release builds.

Mitigation

Only use debug builds.

Remediation

Check inputs such that an exception is thrown when called with invalid values.

Status

BEAM states that, "[t]he input must be validated before reaching the `assert()`". We agree that `assert` is not a good way to reject invalid inputs. However, it is not always obvious where (or even if) inputs have been validated before reaching any given `assert` statement.

To increase code reliability and readability for future audits, we recommend changing many existing `assert` usages into runtime checks that properly return control on errors (perhaps via exception, or return-value as appropriate to the API). For example, a quick `grep` can confirm that the `nMaxOdd` argument from the example above is always currently called with a constant -- but anyone reading the code in the future will have to re-confirm this. For more information, see Remarks on Abstraction, `assert` and Security below.

Verification

Unresolved.

Issue E: Improper Vendoring of Dependencies

Synopsis

Dependencies are not clearly grouped and are scattered over multiple subdirectories, checked in next to regular source files. Some of them have been changed since they were initially checked in. The sources of SQLite are already preprocessed (compacted to very few files), which leads to a smaller repository but makes tracking the version nontrivial.

We expect that updates in the original libraries will be difficult to integrate into the code base, which means security updates in dependencies will likely take a longer time to land.

Impact

Updating dependencies is hindered.

Preconditions

There may be a security issue in one of the dependencies, which goes unpatched because updating them is a difficult undertaking.

Feasibility

Medium. An attacker may wait for a security issue in one of the dependencies to arise and then has some time to exploit it.

Technical Details

This is a partial list of vendored libraries in the source tree:

- `Boost::yas` (for serialization) (in utility/yas). - a few layout changes.
- Equihash POW algo from zcash (at least pow/impl/crypto maybe other code)
- SQLite
- Libuv (utility/io/libuv)
- CHECK: utility/crypto/blake
- QuaZIP (BEAM/ui/quazip)
- Picohttpparser (p2p/picohttpparser)

Mitigation

Update dependencies immediately, even though it is difficult.

Remediation

Implement and use a strict vendoring regime, which means creating a folder “vendor” at the root of the repository and create a subdirectory for each of the dependencies. Modify the files only to pull in updates from upstream. Git submodules can also be used to depend on a vendored library at a particular commit hash.

If changes must be made, a fork of that dependency will need to be maintained. In practice, create a fork, make the changes within the fork, and regularly pull in the changes from upstream. Once complete, the fork can be vendored.

Alternatively, it is possible to not vendor. This allows the build environment to determine the dependency version used. Of course this is only possible if no changes are made.

Status

Most of the 3rd party code is now in a separate “3rdparty” directory. Spotted core/aes.cpp which is a non-BEAM copyright code outside the 3rdparty directory. It still remains somewhat difficult to understand which version of the libraries were imported into the BEAM repository and how bug fixes and updates in the upstream can be imported into the copy in 3rdparty and what modifications are done by BEAM team in these vendored copy (of course, git maintains this history, provided that it also has an unmodified copy checked in as the starting point).

Verification

Partially Resolved.

Suggestions

Suggestion 1: Use Key-Derivation Systematically

Synopsis

If used for key generation, use the one needed for the solving the problem at hand. If a regular nonce or symmetric key is derived, use HKDF (RFC 5869). If the "nonce" for an ECDSA signature to achieve deterministic signatures is derived, use RFC 6979.

If standard algorithms are used when solving problems, it results in security proofs (HKDF is shown to be secure in the random oracle model) or a battle-tested system at a minimum (as in the case of RFC 6979). This also means that for general-purpose keys, HKDF provides security guarantees which are not known to hold for RFC 6979, and thus HKDF should be preferred as a key derivation function.

This particularly applies to

- Callers of ECC::GenerateNonce and ECC::Scalar::Native::GenerateNonce
    - miner/block generation code
    - HKdf (which is not HKDF)
        - Used to make derivations on the keychain in wallet/wallet_db.
    - NonceGenerator in RangeProof::Confidential
    - NodeConnection::SChannelNonce

On the other hand, RFC 6979 should be used for generating the nonces for signatures, i.e. in Signature::Sign.

Mitigation

If you derive a regular nonce or symmetric key, use HKDF (RFC 5869). If you derive the "nonce" for an ECDSA signature to achieve deterministic signatures, use RFC 6979.

Status

ECC::GenerateNonce() has been removed. Since ECDSA signatures are not used, the rfc 6979 based key generation code is removed as well.Instead, the NonceGenerator uses HKDF.

Verification

Resolved.

Suggestion 2: Change type of Loop Variables

Synopsis

Most loop variables do not use `size_t` as their type. This needs to be fixed system-wide. Similarly, the type returned by sizeof() and other size methods of data structures are also of type `size_t`.

In the current implementation of the code, loop variables are sometimes initialized as an `int` and this results in many signed/unsigned variable comparisons.

Eg: `equihash_impl.cpp:202`

```
for (int i = 0; i < lenIndices; i += sizeof(eh_index)) {..}
```

This is because `i` is an `int` and lenIndices is of type `size_t` (unsigned long `int` on my computer which is an x86-64 platform). It is acceptable for loop variables initialized to 0 and going up, however, it is better to get rid of this warning by converting all loop variables to the type `size_t`.

It should be noted that most of these comparison errors are in vendored code (e.g. equihash_impl.cpp). However, since they have been snapshotted into the BEAM code and are being used, it is better to fix problems in that code as well.

Mitigation

Change all loop variables to type `size_t`.

Status

Most of loop variables have been changed to size_t. Those in the “3rdparty” directory is unchanged. There are still some instances of loop variables, outside of “3rdparty” directory which are not of type size_t like in wallet/wallet_db.cpp.

Verification

Partially Resolved.

Suggestion 3: Initialization of Variables

Synopsis

It is recommended to initialize variables. E.g.: code/ecc.cpp:943 (`iBit2` which is initialized in `GetOddAndShift()` - the initialization is inside an `if()` statement’s body, so there is a chance that it may not get initialized and used).

Mitigation

At the declaration of variables, initialize variables with a default value.

Status

The Makefile has been modified to use ”-Wall -Werror” (turn on all warning, warning as error), which would find and flag most of these issues in the future.

Verification

Partially Resolved.

Suggestion 4: Use a Well-Studied Key Exchange Protocol

Synopsis

The BEAM software comes with a new, completely unstudied key exchange protocol. This comes with a significant - and avoidable - risk. There are many great key exchange protocols that currently exist, the most popular of them being TLS 1.3. Using a key exchange protocol that has a decent security proof is best practice in 2018.

Even if you decide to implement a new protocol as a result of existing implementations inhibiting flexibility, it is still strongly advised to use an existing, well-studied protocol.

Mitigation

Use a well-studied key exchange protocol.

Status

The BEAM team responded that TLS does not meet their requirements. However, there are several other well-studied key exchange protocols. They may be particularly interested in secret handshake, which was at least analyzed in the formal model (but not in the computational standard model). For a more theoretical discussion of identity-hiding key exchange protocols, take a look at the paper "Identity-Concealed Authenticated Encryption and Key Exchange" in the proceedings of CCS 2016 (direct link to paper bounces off the paywall).

Furthermore they state that a breach of confidentiality only impacts the confidentiality of transactions, but not the UTXOs; i.e. the adversary learns about transactions, but is not able to steal the victim's money. We want to stress that even though *the worst thing* is prevented, a breach of confidentiality still breaks the promise BEAM makes their users.

Verification

Unresolved.

Suggestion 5: Do Not Implement the Key Exchange Protocol Yourself

Synopsis

Implementing cryptographic protocols independently comes with the risk of making mistakes. To minimize the chances of making mistakes, it is advised to minimize the amount of cryptographic protocols implemented. In practice, this means using an off-the-shelf solution whenever possible.

There are several libraries for key exchange protocols that currently exist:

- Several TLS libraries
- Noise protocol framework (not really a library, more of a protocol construction kit)
- Signal's X3DH
- Implementations for several key exchange protocols, like the secure and fast HMQV protocol, can be found in the Crypto++ library

Prior to creating up with a new and independent implementation, it is advised to explore suitable and compatible existing protocols.

Mitigation

Use an off-the-shelf solution if possible.

Status

This is only possible if you use a key exchange protocol that already has an implementation.

Verification

Unresolved.

Suggestion 6: Limit the Use of Pointer Arithmetic

Synopsis

Pointer arithmetic is used widely in the code, along with `memcpy()` to initialize structures from a binary blob. It may be a good idea to use proper serialization/deserialization functions from/to binary blobs as as initializing structs with `memcpy()` relies on the memory layout assumptions, which may break between compiler/platform versions.

A selection of instances in the code::

1. `wallet/keystore.cpp`, pointer arithmetic and `memcpy()` is being used to initialize structures.
2. `ECC::InitializeContext` (in core/ecc.cpp), casts a buffer to type Context

Mitigation

In the reported case, `ECC::NoLeak` is just a wrapper around a value of `type T`. For more complicated structures, initializing via an initialization function to initialize members of a struct/class is desirable. Also, instead of relying on pointer arithmetic, a structure representing the data could be defined and serialization/deserialization functions written for specific data representation in the memory.

Status

`wallet/keystore.cpp` was removed from the codebase. The cast on `ECC::InitializeContext()` seem fine as this is at the initialization phase and there is no data de-serialization being performed there.

Verification

Resolved.

Suggestion 7: Use Division and Multiplication for Converting Bit/Byte Counts

Synopsis

Conversion between a bit count and the corresponding byte count happens fairly often. The usual conversion formula from a byte count to a bit count is “nBits = nBytes * 8”. However, the code of the BEAM project usually uses a bit shift by three positions. While this is arithmetically equivalent (on known machines by the auditors at a minimum), it is much less clear what the intention is.

It appears to be an exercise in premature optimization, because multiplication is more expensive than bit shifts. However, the compiler should make a bit shift out of “someInt * 8” itself, and even if not, the difference is very few cycles and unlikely to have a substantial impact on performance. In practice: measure then optimize.

Mitigation

Use multiplication by eight to convert from bytes to bits and division by eight to convert from bits to bytes.

Status

While we acknowledge that this does not have direct impact on security, we want to stress the indirect impact that results from the more difficult maintainability that comes with quirks like this.

Verification

Unresolved.

Suggestion 8: Custom Curve Crypto Should Only be Used when Really Needed

Synopsis

In `core/ecc*`, much of `secp256k1` is wrapped and/or exposed. For example, it provides a high-level API for operations on group elements in multiple formats. This layer adds a source of possible mistakes. In addition, bugs fixed in the original library will have to be manually adapted to `core/ecc*`.

We are aware that the custom cryptographic code is needed for both Mimblewimble and Bulletproofs. However, in situations where these special features are not used and off-the-shelf `secp256k1` is sufficient, it is the safer alternative. One example for where this applies is the key exchange.

Remediation

Use direct calls to `secp256k1` in instances where only the default generator of the curve is needed.

Status

The BEAM team chose not to change this, as it is not critical and they have high confidence in their custom curve implementation.

Verification

Verified: Unchanged.

Remarks on Abstraction, 'assert' and Security

In relation to several of the Issues and Suggestions we offer some further explanation.

In software development Abstraction means ignoring implementation details: small API surfaces, catching invalid data at every abstraction boundary, and not leaking implementation details across these boundaries. This allows much easier reasoning about the behaviour of large code bases.

This also aids security, because to verify the security of some larger piece of code, reviewers only need to look at small pieces at a time to verify that there no security issues. Otherwise, auditors and developers have to keep track of a lot of state and edge cases that may occur, and sooner or later issues will come up when they are unable to track sufficiently.

The larger the API surface, the more possibilities of something going wrong are required to be tracked. If there are abstractions that don't catch invalid inputs (which often result in security issues), reviewers have to find every user of that abstraction and try to verify that the abstraction is only used with valid inputs.

Requiring that every abstraction checks the validity of its inputs before operating on them means that even if something went wrong, nothing bad happens, but an error would be returned. If that error happens in tests, something will return this error, the test fails, and the reviewer will know what's going on. However, in a crash that happened due to a failed assert, the reviewer would not know what was happening. But, more importantly, if the same thing happens in production, we don't execute the function on invalid inputs, possible leaking secret, but we simply return an error. This error can be shown to the user, or simply sent to the dev team so they can look into how that happened.

We can't be sure that we always do the right thing (i.e. operate on valid inputs), because when we get passed invalid inputs there is no right thing. But we *can* make sure that we don't do the wrong thing (i.e. we abort execution on invalid inputs).

To give an example, right now `Hash::Processor` is used in a lot of places, and verifying that *under no circumstances* it is used without proper initialization is nearly impossible. To be confident that we never use an uninitialized hash function, an assert is not enough. As discussed, these are helpful for debugging, but this is about checking both the validity of the inputs and wether the internal state is sane.

Similarly, it would be good if `MultiMac::FastAux::Schedule` properly checked that the `nMaxOdd` parameter is valid (i.e. odd). Otherwise, if some caller passes an even value in production code, it will silently do what it does - on illegal inputs. This may result in anything from crashes to leakage of secret data. Saying "it's the responsibility of the caller" is not enough, because your program is too complex to effectively rule out the possibility of something calling it with an even `nMaxOdd`. Checking the validity of inputs isinexpensive and needs to be done in every function. Of course, there are exceptions to this rule, but these are *exceptions* and not the norm.

Note that **Issue C** was not about "use TLS". It was about structuring the code similar to the way TLS does it. You don't have an RPC protocol, a key exchange protocol and a secure channel protocol. You have an RPC protocol that also does a key exchange and encryption of RPC messages. This is a huge abstraction, where the state of key exchange, secure channel and RPC are all mixed into one area of code. Analyzing the security of this is very difficult, because there are a lot of moving parts. If this would be separated into a key exchange module, a (possibly off-the-shelf) secure channel module and then a (possibly off-the-shelf) RPC module, with each of these modules having only access to their own state, the confidence in the result of the audit would be much higher, because if the auditor has to keep track of less moving parts at the same time, the chances that they miss something are much lower. Reasoning about the security and functionality of the code is not only important at audit time since this codebase will further evolve and you as developers have to think about this as well. Furthermore, as your team grows, developers new to the project will also have to understand the code base well enough to contribute to it, and if every second change they make breaks something at some completely different part in the code base, this will severely hinder your progress. Also note that sometimes these will be subtle breakages that work in the average case but introduce security-critical edge-cases.

We hope this helps you understand why many of the issues and suggestions in this report were about abstractions and asserts.

Recommendations

We recommend that there be further analysis on the unresolved and partially resolved *Issues* and *Suggestions* stated above and that they are addressed as soon as possible and followed up with verification by the auditing team.

Additionally, we recommend that the code readability continues to be improved as the codebase grows to facilitate easier code reviews and external contributions in the future.

The codebase can be further improved by properly vendoring dependencies, increasing test coverage report generation and generally increasing the number of comments in the code.

All of these changes would reduce the risk of code errors and therefore security vulnerabilities.

Finally, we recommend that additional security audits be conducted on future development releases to ensure that any potential issues and vulnerabilities are identified, addressed and verified.

Appendix 1: Activity Log

These are notes from the reviewers about their activities during the code audit. They detail the approach and investigative activities undertaken. All issues found are listed in the report. This is just for the purposes of transparency and could be helpful for another auditor to understand the evaluation activities.

Log 1:

2018-10-01

Built the code from source from git commit hash: 6ce6a409c839ce7cebb0aa1a7cc1d1154458a07e without any problems.

Some immediate goals:

1. Learn about mimblewimble.
   https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.txt
2. Learn about range-proofs.
3. How the graph of transactions are exposed/not-exposed etc.
4. How the UTXOs work in BEAM.
5. Greg Maxwell's "Confidential Transactions" proposal which seem to be the basis of MW.

2018-10-02:

Modified cmake file to add -fsanitize=XXX options to the CXX_FLAGS.

- -fsanitize=address and running the tests crash some tests, but I couldn't pinpoint whether they are because of actual memory issues. It is most likely not because of real issues.
- -fsanitize=undefined and running tests also didn't show anything.

This didn't yield much issues.

2018-10-03 - 2018-10-04:

Read the intro.md and correlate with the code. Read ecc.cpp, ecc.h

2018-10-05:

ecc.cpp: Recommend using size_t for loop variables that index into an array. Several instances of it in ecc.cpp.

2018-10-31:

p2p/* - protocol between the peers. Looked at the handshake. At first it looks like a naive DH handshake. It looks like it is not, as it derives the class from secp256k1-zkp. Need to check whether the handshake is authenticated.

2018-11-04:

Started looking at the proof of work algorithm. It is the equihash algorithm used in Zcash. Spent time reading the equihash paper and the Zcash python implementation.

2018-11-05:

Equihash implementation underneath is using Zcash's code unmodified.

2018-11-06:

Read the bbs.cpp/h files again and the bbs_client.cpp. It looks fine.

Ran unit tests (make test) with -fsanitize=address, -fsanitize=leak turned on (one by one). A couple of tests fail with -fsanitize=leak (for the node test in particular). Sanitizer reports a memory leak of 40 bytes. But I am unable to pinpoint where the problem is. Valgrind reports some other problems, not a memory leak. Test fail under valgrind too. But it passes when run without valgrind or without -fsanitize=leak.

Most loop variables do not use size_t as their type. This needs to be fixed system-wide.

There are many signed/unsigned variable comparisons.

Eg: equihash_impl.cpp:202

for (int i = 0; i < lenIndices; i += sizeof(eh_index)) {..}

This is because i is an int and lenIndices is of type size_t (unsigned long int). For loop variables initialized to 0 and going up, it is okay, but it is better to get rid of this warning by converting all loop variables to the type size_t.

It needs to be noted that most of these comparison errors are in code imported from elsewhere (like equihash_impl.cpp).

It is also a good idea to initialize variables. Eg: code/ecc.cpp:943 (iBit2 which is initialized in GetOddAndShift() - the initialization is inside an if() statement's body, so there is a chance that it may not get initialized and used).

MMR: still very confused about it. Reviewing the code and the telegram messages/wiki pages.

2018-11-07:

Mostly understood MMR but still can't grok the code. Re-reading the wiki pages.

2018-11-08:

wallet/keystore.cpp:

Lots of pointer arithmetic. Lines 108, 109 uses memcpy() to initialize classes. It may be a good idea to use a class initialization method instead? (same comments for other memcpy operations in the same file).

read_keystore_file():L89 - do we need to close the file? Won't AutoClose destructor automatically handle it?

2018-11-12:

handshake/key exchange: It looks okay to me. It does authenticated handshake, however, it looks like a trust on first use system. I couldn't find a way to associate IP with the privateID that a malicious node can use to identify a node..

Log 2:

2018-10-02 - 2018-10-09:

I've started to get familiar with the code (especially core/) and to investigate the cryptographic operations. I took a look good at the Bulletproofs paper to grok the proofs, so I can verify the implementation is correct.

2018-10-11:

I read through most of the Wiki docs. Observations:

- `Hash::Processor.Write(const char*)`
  - writes null-terminated strings. This encoding makes no difference between `(a||b)||c` and `a||(b||c)`. I suggest length-prefixing the strings and omitting the `null` byte.
  - Does not define the encoding the incoming data is in. I'm not yet sure whether that is a problem.
- In the Transaction Kernel
  - I'm not yet sure whether the m_Excess public key is the public key of the UTXO of this transaction, or the one that gets consumed.

With this new background, I looked at the code again:

- In `Node::Miner::OnRefresh`, the block nonce is calculated as `H(minerID || KDFSecret || idx || height)`, with `idx` being a parameter to the function. That means that if the function is called with the same parameters before the height changes, the same nonce will be generated.
  Also, this effectively is a key generation, and for this a key derivation function should be used instead of a plain hash. I suggest using HKDF. The HKDF secret also should not be the same as the ECC KDF secret - though they may be derived from the same secret.

2018-10-12:

- Read through the Full System State wiki docs
- The miner code generates the block nonce by hashing a few values, including the ECC::Kdf secret. I suggest (1) not simply hashing, but using HKDF and (2) not use the ECC::Kdf secret, but a secret specifically for deriving other nonces. I discussed this in the Telegram channel.
  - They immediately pushed a commit removing the ECC::Kdf secret. This makes the nonce pretty predictable, not sure yet whether this is a security issue.
- The miner code interleaves test code and production code (if Rules::get().FakePOW{...}), which is not very clean and "less obviously correct", i.e. while not necessarily wrong, it's easier to slip in mistakes this way, or enable ROP-style attacks (if the attacker somehow manages to jump into test code).

2018-10-13:

- Found out that the >> operator of the hash processor modifies the internal state of the hash and finalizes it. After finalization the internal state of the hash function is zeroed and you need to reinitialize the hash before further writing into it.
  - Example: MultiMac::Prepared::Initialize (core/ecc.cpp:690) calls Generator::CreatePointNnz(val, hp) (core/ecc.cpp:511) (which finalizes the hash) in a loop and writes to it in every iteration. There are more instances.
  - Possible fixes: re-initialize the hash in the loop or copy the hash and finalize the copy
  - Also happens in Generator::CreatePointNnzFromSeed.

- Correction: the Hash::Processor::Finalize method refeeds the hash value into an uninitialized hasher. Discussed the issue with Vladislav and came to the conclusion that it's best to finalize a copy of the hasher, so we can keep writing into the original one.

2018-10-23:

- Throughout the code base, integer (non)zero checks are often performed by "if (someInt)" or "if (!someInt)". I suggest using the correct comparison operator (i.e. <, >, ==, !=) with zero.
    - Example: If (someInt) -> if (someint != 0)
    - Exhibit: Merkle::DistributedMmr::Impl::LoadElement, SaveElement
    - Hm, looks like that's best practice in C++, nevermind
- In DistributedMmr::Append
    - why is pBuf of void* but then casted to Hash*?
    - Why cast *Hash to *Key?
        - It's pointer arithmetic to read a Key after a couple of Hashes.

2018-10-24 - 2018-10-25:

- My current quest is to check that the MMR (both local and distributed) are correct
- DistributedMmr makes heavy usage of the NodeDB, so I'm reading through that
    - In NodeDB::Transaction::Rollback the exception is caught but not handled (there is a TODO comment).

2018-10-26:

- I'm still figuring out what actually happens in Dmmr/DistributedMmr. I asked in the Telegram what the member variable DistributedMmr::Impl::m_pNodes is about (an array), because it is iterated over and read at several positions. This variable is used extensively in the algorithm for searching in the distributed mmr, so it's important.

2018-11-01:

- In ECC::GenerateNonce there is a loop that will always return on the first iteration, because nonce_function_rfc6979 always returns 1.
- There are two data types called uintBig, one is 128bit and the other is 256bit. In core/ecc.cpp, the 256 one is used.

2018-11-02:

- The docs state that one reason for not simply using secp256k1[,-zkp] is that they need 131 (according to the docs) generators and the library only provides two. Looking into the computation of generators.
    - Generators are computed from a static seed of both (a) the string "Let the generator generation begin!" and (b) the secp256k1 generator g.
    - I didn't find checks that the computed generators for sufficiently large subgroups
        - Since secp256k1's cofactor is 1, every element of the group is a generator for the whole group. So no small subgroups!

2018-11-05:

- Looking at the generator generation code again, I am not sure whether the points computed in ECC::Generators::CreatsPts have too much structure that might simplify discrete-log computations.

    - These are not the generators, but values that make exponentiation of the generator faster.
- In general I find the Oracle data type very difficult to reason about, because every read also modifies the internal state. This is supposed to be a security feature, such that successive reads return different values; but it's not obviously clear whether there are multiple paths that lead to the same value.

2018-11-12:

- Move items from the audit log to the Issues and Suggestions sections
- Discussed Key Exchange questions on Telegram