Least Authority
PRIVACY MATTERS

Atomex: Core Library + Desktop Client

Security Audit Report

Tezos Foundation

Final Report Version: 24 September 2020

Table of Contents

Overview

Background

Tezos Foundation has requested that Least Authority perform a security audit of the Atomex core library and desktop client. Atomex is a hybrid exchange based on atomic swap technology and multicurrency hierarchical deterministic (HD) wallet.

Project Dates

- **April 29 - June 3:** Code review *(Completed)*
- **June 5:** Delivery of Initial Audit Report *(Completed)*
- **September 21 - 23:** Verification *(Completed)*
- **September 24:** Delivery of Final Audit Report *(Completed)*

Review Team

- Nathan Ginnever, Security Researcher and Engineer
- Dylan Lott, Security Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Atomex core library and desktop client followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:
- Atomex client core library: https://github.com/atomex-me/atomex.client.core
- Atomic client for OS Windows: https://github.com/atomex-me/atomex.client.wpf

Specifically, we examined the Git revisions for our initial review:

Atomex.client.core: `2cf279bfd4202e90b9534b0f797b428e9c7e3d87`

Atomex.client.wpf: `cab4af61379d13adab90b65d526187083a799f91`

For the verification, we examined the Git revision:

Atomex.client.core: `6c987aca51b760c900c0ca87557ccbd920de333f`

Atomex.client.wpf: `d34fdec28e334c88b3bc21e9f55af66e87ba1d93`

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:

- Attacks that impacts funds, such as draining or manipulating of funds;
- Mismanagement of funds via transactions;

- Secure communication between the nodes;
- Proper management of encryption and signing keys;
- Vulnerabilities within each component as well as secure interaction between the contracts and network components;
- Correctness of the implementation;
- Adversarial actions and protection against malicious attacks on the network;
- DoS/security exploits;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

While the functionality in both the Atomex core library and desktop client was closely observed and evaluated for security vulnerabilities, the core library contained the majority of the code while the desktop client contained GUI code that called functions on the core library. As a result, all of the issues reported by our team correspond to the code found in the core library repository and we identified no security vulnerabilities in the client GUI code.

We also note a few issues as it relates to dependency concerns, including BouncyCastle's Ed25519 fork (Issue B), NBitcoin taking keys as byte arrays, which does not allow them to be in secure memory (Issue A), and LiteDB's encryption (Issue C).

Design

The overall design of the core library and desktop client demonstrate an effort to address various security considerations, but require modifications to ensure they effectively achieve the intended security goals. We recommend correcting some basic errors that have been identified in the cryptographic components of the system (Issue B; Issue C; Issue D). We also note a few issues as it relates to dependency concerns, including BouncyCastle's Ed25519 fork (Issue B), NBitcoin taking keys as byte arrays, which does not allow them to be in secure memory (Issue A), and LiteDB's encryption (Issue C). We also recommend some further improvements to the design such as providing a clear method for backing up the wallet private keys (Suggestion 6) and relying less on third-party services to maintain connections to the blockchain for the wallets by standing up nodes and infrastructure to support the Atomex application (Issue G) or centralized APIs (Suggestion 2).

With the atomic swaps, we focused most of our efforts on checking the security and handling of the data, as well as the safety and structure of the Atomex swap setups, refunds, and claims given that swaps themselves are a well-understood area of cryptography. Creating an application that relies on multiple chains provides a challenge as infrastructure for each chain must be maintained and connected to. Events like reorgs (Issue E) or lost state during a swap (Issue F) provide extra complexity to creating a secure atomic swap exchange.

Code Quality & Documentation

We found the code in both the core library and desktop client to be well-organized with considerable reasoning behind project structure and coupling. The code also appears to follow and adhere to C# styles.

However, the desktop client and the core library were insufficiently commented, particularly given the large size of the code base, which resulted in difficulty in building and testing the code (Suggestion 8).

While the Atomex desktop client included some description of the operating system and build dependencies, in addition to a basic build description, information on the proprietary dependencies needed in order to build the desktop client, such as Microsoft libraries, was missing. The Atomex core library also failed to provide sufficient information and documentation on the instructions to build, install, and test the code and the required operating system or version of Windows. Furthermore, the lack of supporting documentation for the functionality of the core library and the desktop client made it difficult at times to deduce the role of a function.

Although we found some documentation included on the Atomex site, including how to Stake Tezos using the Atomex wallet and an FAQs section, the project would significantly benefit from additional documentation including a well-written README, a general description of the implementations, how certain problems are solved, the intention behind the design and expected behaviors, description of the dependencies (e.g. Better-Call-Dev, Baking-Bad, and Etherscan), and certain security limitations as described in Issue A (Suggestion 8).

Test Coverage

There is a sufficient amount of test coverage for the Bitcoin code, however, we recommend more comprehensive coverage of tests that check the error paths. Additionally, for the Ethereum side of swaps, we found that some tests were commented out and that critical parts of the code base were missing test coverage. In particular, the core library would benefit from additional test coverage and a more robust test suite (Suggestion 3).

Specific Issues

We list the issues 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: SecureBytes Does Not Prevent the Copying or Paging Out of Key	Partially Resolved
Issue B: BouncyCastle's Ed25519 Fork is Not Well-Maintained	Unresolved
Issue C: LiteDB's Encryption is Broken and Unreliable	Resolved
Issue D: Key Derivations from Passwords are Insecure	Resolved
Issue E: Default Block Confirmation Does Not Account For Blockchain Reorgs	Unresolved
Issue F: Dropped Swaps Are Not Re-initiated or Re-accepted	Unresolved
Issue G: API Shared Resources May Be Unstable	Unresolved
Issue H: Hash Time Locked Contract (HTLC) Preimage Secret is Not Stored Safely in Memory	Invalid
Suggestion 1: Use Certificate-Pinning for all Atomex API Endpoints	Unresolved

Suggestion 2: Explore Paths to Reduce Single Points of Failure	Resolved
Suggestion 3: Increase Test Coverage	Partially Resolved
Suggestion 4: Add Hardware Wallet Support	Partially Resolved
Suggestion 5: Remove Unused Code	Partially Resolved
Suggestion 6: Provide a Method for Exporting Private Keys	Resolved
Suggestion 7: Look for Memory Encryption Libraries for C#	Partially Resolved
Suggestion 8: Improve Documentation	Unresolved

Issue A: `SecureBytes` Does Not Prevent the Copying or Paging Out of Keys

Location

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Common/SecureBytes.cs

Synopsis

`SecureBytes` is a class that stores binary data in a way that aims to mitigate against an attacker reading keys from memory. While the technique is not problematic, the functions that take the keys as arguments are generally not compatible with this approach. Therefore, this approach is not as effective at mitigating attacks as intended. In addition, we identified various places in the code where no security measures are in place to store private keys, but plain `byte[]` is being used instead.

Impact

If the attacker manages to circumvent this current mitigation or is able to read private keys from insecure `byte[]` storage, they may gain access to wallet keys.

Preconditions

The operating system would need to page out the areas in memory that contain the key byte slices to a pagefile and the attacker would need access to the file. The attacker would also need to be running code on the device of the user. Depending on the security habits of the user and the design of the operating system, it may suffice if the code of the attacker runs with user privileges.

Feasibility

The attacker would need to be able to analyze the pagefile or memory dump. This requires some knowledge in the field of computer forensics.

Technical Details

C# has managed memory, which means that it is much easier to write memory-safe code. Memory safety means that the possibility of double free, buffer overflows, and similar attacks is drastically reduced through the use of a runtime. The memory is managed by the common language runtime (CLR). The CLR may move or copy memory around, without the developer being able to prevent it, so perhaps surprisingly, memory-safety introduces issues around the secrecy of the contents of the memory. This means that for regular byte arrays, it is not possible to safely overwrite a key with zeroes. Additionally, memory may be paged out to the pagefile at any time. This means that regular values stored in memory may be written to disk.

To address this issue for storing passwords in memory, Microsoft has added the `SecureString` class, which stores a string such that is neither copied nor paged out. The Atomex team uses this class to write `SecureBytes`, which securely stores binary data (such as keys) encoded as hex in a `SecureString`. Although this is not a poor practice, the issue remains that all functions that use these keys for cryptographic operations (e.g. key derivation, encryption or signing functions) are not able to use `SecureBytes`.

Remediation

The libsodium library provides functions to allocate free memory that can not be paged out. There are bindings to that library in C#, such as NSec. We recommend using one of these to allocate unmanaged memory before and overwrite it with zeros as well as free it after use.

Status

The Atomex team has changed `SecureBytes` to use memory allocated by libsodium. This mitigates many of the possibilities to leak keys through memory dumping attacks. However, several cryptographic operations still operate on plain byte arrays requiring keys to be converted to the less secure format, thus creating an opportunity for moments of vulnerability. We recommend that further effort is invested in finding and using, or possibly even implementing, operations that currently require insecure memory to function, e.g. the BIP32-Ed25519 library.

Verification

Partially Resolved.

Issue B: BouncyCastle's Ed25519 Fork is Not Well-Maintained

Location

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Cryptography/BouncyCastle/Ed25519.cs

Synopsis

The BouncyCastle `Ed25519` fork is out of date. Since the Atomex fork was made, more security checks have been added to the original BouncyCastle code, specifically a low-order point check for all keys. Furthermore, the code that has been forked is undocumented. As a result, given that BouncyCastle is central to the security of the core library, it is critical that the code be regularly updated and maintained.

Impact

Issues in the cryptographic library used to create and verify signatures may be able to trick a wallet into accepting invalid signatures, which are used to verify the validity of transactions.

Preconditions

Two preconditions are possible. First, the attacker may find a hole in the changes made to the `Ed25519` class by the Atomex team or, second, a security vulnerability in BouncyCastle is found and fixed, but not merged by the Atomex team.

Feasibility

If the preconditions are met, an attacker with decent cryptographic knowledge would be able to attack users. Significantly, in the case that a fix to a vulnerability in the BouncyCastle library is not merged into Atomex, that vulnerability would be publicly visible.

Technical Details

BouncyCastle is a popular cryptographic library for Java and C#. However, the C# version appears to be completely undocumented. The Atomex team has copied the class providing `Ed25519` signatures out of the BouncyCastle project and made some changes.

The reason or purpose for the changes made and whether they are secure appears to be neither documented nor discussed. Through extensive study of code, it becomes clear that the changes were made because the API provided by BouncyCastle did not allow implementing BIP32-Ed25519 [KL17], a scheme for hierarchical deterministic (HD) wallets. Specifically, BouncyCastle only allows using the 256-bit seed value as a private key, but not the 512-bit key (called the *expanded* key in BIP32-Ed25519), which is required by BIP32-Ed25519.

Forking and maintaining a library, particularly one that includes cryptography, requires a considerable amount of work and responsibility. Instead of modifying a library to match the project requirements, a more pragmatic approach would be to choose a better suited library. Since BouncyCastle does not appear to meet other requirements of the library, the Atomex team implemented the `SecureBytes` class to protect keys in memory. However, BouncyCastle is not able to work with this type, thus making the keys they try to protect vulnerable.

Remediation

Given that maintaining a fork requires significant effort, instead of using BouncyCastle, we recommend using the NSec cryptographic library to implement BIP32-Ed25519, in which the API allows deriving secret (expanded) keys from a seed as a secret key for signing. While the change to support BIP32-Ed25519 keys is considerable, it is significantly smaller than the current changes to BouncyCastle. The library also resolves the issues around protecting keys in memory against getting paged out by exposing the respective libsodium functions. In addition, using NSec provides free secure memory.

Status

Considering the current unavailability of a cryptographic library which immediately implements both secure key handling and signing using Ed25519 with keys that are derived hierarchically deterministically according to to BIP32-Ed25519, the Atomex team has decided to maintain BouncyCastle by tracking and including changes made upstream. However, the Atomex team did not provide a clear process for this undertaking. Instead, they implemented several changes to the files copied out of the BouncyCastle project, including having the code align with their own code style (i.e. whitespace, `var`-style declarations, curly braces around single-line for loops, etc.). Furthermore, the new code added by the Atomex team is scattered throughout the file, instead of kept mostly separate, which would have helped comparability. As a result, we recommend that in the absence of a clear process for tracking and maintaining the remediation be implemented in order to resolve this issue.

Verification

Unresolved.

Issue C: LiteDB's Encryption is Broken and Unreliable

Location

https://github.com/atomex-me/atomex.client.core/tree/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/LiteDb

Synopsis

LiteDB is used to store wallet data on disk. The database has an encryption mode that is used by the Atomex team. The security of the encrypted mode of LiteDB is severely flawed and can not be considered secure.

Impact

An attacker may get access to the wallet data, including but not limited to, the secret values used in the atomic swaps. The attacker may also change the client state by modifying the encrypted data in the wallet. Combined with the vulnerabilities in Issue D, this may also result in leakage of the wallet keys.

Preconditions

The attacker would need access to the encrypted LiteDB database. This is possible for any software running on the disk with user privileges.

Feasibility

The attacker would need some understanding of applied cryptography at the same level as most blockchain engineers. The attacker could catastrophically attack the encryption of LiteDB, compromising the database in its entirety.

Technical Details

The encryption of LiteDB is based on AES-ECB, which is known to be vulnerable and should not be considered secure. As a result, it should not be relied on for storing wallet data on disk. Furthermore, the encryption is not authenticated, allowing the undetected modification of data. Additionally, the key derivation process used to derive keys is PBKDF2 with only a thousand iterations. LiteDB also does not specify which underlying function to use so the default of HMAC-SHA1 is used (SHA1 is broken and should no longer be used). As a result, this method does not provide sufficient protection against brute force attacks.

Remediation

Rather than relying on LiteDB's encryption, encrypt the data manually before storing it. To do so, do not use a password but an already strong key K instead. That key may be derived from a password, which we discuss further in Issue D.

To store an item **I** at address A in the database:

- Compute the shadowed address
    - `A’ = HMAC(K, “addr” || A)`
- Compute the encryption key for item **I**
    - `K_I = HMAC(K, “key” || A)`
- Pick a 192 bit random nonce
    - `N = rand(24)`
- Encrypt the item
    - `I’ = secretbox_encrypt(K_I, I, N)`
- Store I’ at A’
    - `DB.Set(A’, N || I’)`

To retrieve an item at address A in the database:

- Compute the shadowed address
    - `A’ = HMAC(K, “addr” || A)`
- Compute the encryption key for item **I**
    - `K_I = HMAC(K, “key” || A)`

- Get the nonce N and the encrypted item I’ from the database
    - `N || I’ = DB.Get(A’)`
    - where N is the first 24 bytes of the results and I’ is the rest
- Decrypt the item
    - `I = secretbox_decrypt(K_I, I’, N)`

Use the `secretbox` functions provided by a library that provides bindings to libsodium, such as NSec. This algorithm not only provides confidentiality, but it will also detect if an attacker attempts to modify the data.

Status

The Atomex team implemented manual encryption of all stored items based on AES256-GCM.

Verification

Resolved.

Issue D: Key Derivations from Passwords are Insecure

Location

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Wallet/UserSettings.cs#L17

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Common/SessionPasswordHelper.cs

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Wallet/HdKeyStorage.cs#L25

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Cryptography/Aes.cs#L12

Synopsis

There are several locations where keys are derived from passwords throughout the code base. However, we do not see a consistent strategy as different approaches and parameters are utilized in different places. None of the parameters chosen are sufficient against brute force attacks.

Furthermore, keys derived from passwords are used both for database encryption and secure key storage. This constitutes a form of key reuse, which can lead to cryptographic vulnerabilities.

Impact

An attacker may be able to brute-force the encryption keys of the database and secure key storage or perform related-key attacks on AES itself. In each case, they would be able to learn at least parts of the plaintext data.

Preconditions

The attacker would need access to the encrypted wallet. This is possible for any software running on the disk with user privileges.

Feasibility

For the brute-force attack, the attacker would need sufficient computational power. Due to the insufficient technique currently being used, this would be inexpensive, especially considering the potential rewards. For related key attacks, the attacker needs profound cryptanalysis knowledge.

Technical Details

Most of the key derivations from passwords in Atomex use PBKDF2, but with varying iteration counts. LiteDB uses an iteration count of 1024 (e.g. very low compared to the 100,000 iterations that were used by LastPass in 2011), and before the user password is passed to LiteDB, it is iteratively hashed with SHA256 ten times. The encryption and key derivation of LiteDB is discussed in Issue C, yet the purpose of this is unclear. One possibility is that should LiteDB corrupt the key, the original password is not affected. However, the issue remains that if the attacker knows the tenth hash, it is still possible to brute-force the password.

The AES class uses a default iteration count of 52,768, however, when the class is used in the classes `UserSettings` and `HdKeyStorage`, it is overridden with 1024. This value is much too low.

In general, PBKDF2 has the weakness that it is parallel. As a result, memory-hard hash functions should be used for deriving keys from passwords. They significantly increase the cost of parallelization.

Furthermore, in some instances keys derived from passwords are reused while in other instances the derived keys are different but related, because the only difference is the iterated hash before the derivation. Instead, a single key should be derived from the password, from which further keys are derived using a regular key derivation function like HKDF.

Remediation

Derive a single key K_{pw} from the password using Argon2id or Balloon Hashing. To choose the parameters, follow the recommendations in Section 4 of the RFC draft for Argon2. As a guidance, memory sizes of 32 and 64 MB are common today. For each part of the system that requires a key (e.g. database encryption or HdKeyStorage), derive a key from that using `HMAC-SHA256`. Use K_{pw} as the key and a label describing the user of the key as the message. For example, the key for database encryption K_{db} would be computed as follows:

$$K_{db} = \texttt{HMAC-SHA256}(K_{pw}, \text{“database encryption”})$$

Status

The Atomex team implemented our recommendation and, as a result, the Argon2id is used to generate a key from the user's password. Furthermore, the key is calculated from the main key using `HMAC-SHA256` for any encryption.

Verification

Resolved.

Issue E: Default Block Confirmation Does Not Account For Blockchain Reorgs

Location

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Blockchain/Helpers/TransactionConfirmationHelper.cs https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Blockchain/Ethereum/EthereumTransaction.cs#L20

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Blockchain/BitcoinBased/BitcoinBasedTransaction.cs#L18

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Blockchain/Tezos/TezosTransaction.cs#L19

Synopsis

Blockchain reorganizations are an issue that all applications must deal with. The default block confirmation is set to one block in the Atomex core library, regardless of the underlying chain in consideration. If the wallet is performing actions, and a chain reorganization occurs at a critical time of the atomic swap, there could be a need for the wallet to recover and to rebroadcast a transaction or retry a swap.

Preconditions

A wallet must be connected to a node that includes their transaction in a forked chain, and after one block or more, the fork is reorganized to the canonical chain while transactions and actions taken by the client have assumed that transactions are confirmed.

Feasibility

Blockchain reorganizations are more common on some chains than others. Given that they are frequent in Ethereum due to low block time, this is likely to happen. Ethereum refers to these frequent forks as ephemeral forks. While these forks are generally short, it may be possible that they are extended unexpectedly.

Technical Details

Many checks before performing actions for swaps or other transactions are checked against a boolean stored on the transaction called `isConfirmed`. This boolean is set to `true` if the transaction has been included in the blockchain and is greater than `DefaultConfirmations`. With a default of one block of confirmation, there are cases that a client is connected to a node that has not discovered a fork with a heavier amount of observed work to it from other peers. If the state of a node the wallet is connected to is not derived from the canonical chain, there is a chance that a transaction will be made and then later removed due to a reorganization. The client database may get updated with a state that is now incorrect and needs to be reconciled with the new fork. This might lead to very different swap outcomes (e.g. if a timeout occurred on one branch and no time out on another).

Remediation

One approach would be to handle `DefaultConfirmations` differently for any chain and adhere to the recommended minimum of blocks to wait before considering a transaction to be confirmed. The problem with this approach is that there will be a longer delay on all transactions, which may affect the usability of the Atomex swaps.

The other approach is to add checks to the core codebase. These additions would be checks for reorganizations, database inconsistencies, and failed transactions over a longer period of time. The issue with this approach is the added complexity to the implementation, however, this may be the safest route and can be limited in scope to swaps within the default time span a swap may live, currently set at a default time span of a maximum of 10 hours.

Status

The Atomex team has responded that the implementation of additional checks and tracking for reorganizations is in progress. However, this effort was incomplete at the time of this verification.

Verification

Unresolved.

Issue F: Dropped Swaps Are Not Re-initiated or Re-accepted

Location

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Swaps/SwapManager.cs#L451

Synopsis

It is unclear what occurs in the case that a swap is lost by the client software. There is a function in the `SwapManager` class called `RestoreSwap` that appears to address this occurrence, as it checks the state of the swap and will attempt to act on it accordingly. If the swap is active, meaning that it has not yet reached the default payment timeout, the code will do nothing as it is not implemented.

Impact

This is a low priority issue as these swaps have not yet had payments broadcast to them so no funds are at risk of being lost. Therefore, this would only lead to inconsistencies in the swap clients when a swap is lost and needs to be retried before payments are made. A client may be stuck on a particular swap until the swap timeout without the ability to continue.

Preconditions

An initiator client creates a swap and loses it before sending it to their counterparty and now must resend the swap, or a recipient of a swap loses their swap and should now respond when it is recovered.

Feasibility

This is not highly feasible as the swap should be lost before any broadcasts of payments are made.

Technical Details

If a swap payment has not yet been broadcast and the time is still within the swap's timeout timestamp, a method should exist for recovering the swap or re-initiating it by sending the swap to the recipient or by re-accepting a swap sent to the client. Currently, the implementation does nothing and leaves the swap there until the end of the default payment time where it would then be removed.

Mitigation

Implement the missing logic for re-initiating swaps in the case that payments have not yet been broadcast to the chain.

Status

The Atomex team has responded that this is a low priority task and have not addressed the issue at the time of this verification.

Verification

Unresolved.

Issue G: API Shared Resources May Be Unstable

Location

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Blockchain/Ethereum/EtherScanApi.cs#L24

Synopsis

On the Ethereum side, Atomex uses APIs for services such as Etherscan or Infura. These services have rate limits that can be exceeded if clients overwhelm them with requests. And when these services are overwhelmed, they have been known to be faulty and return timeouts.

On the Tezos side, services like tzStats, or Baking Bad are used which do not enforce limits on the number of calls or the amount of data one can query from the API. However, spam protection measures that limit the number of connection attempts and HTTP calls over short time-frames are in place.

Impact

If the client wallets rely on a single blockchain service account and that account is open to take requests from anyone, then the service could be rate limited and shut down or overwhelmed from general or attacker traffic. This would block all wallets relying on that service from communicating with the blockchain. This could come at opportune times for an attacker during the lifespan of a swap.

Preconditions

A large amount of traffic is sent to the services that the wallets rely on to interact with each chain, causing the service to become overwhelmed and shut down for a period of time.

Feasibility

The stability of the blockchain API services has been known to be inconsistent for large amounts of traffic making it likely that the use of the third-party service will experience failed requests.

Technical Details

The Etherscan API endpoint used has a documented rate limit. The Infura documentation mentions that all accounts have a rate limit and list how they score these limits. The recommended solution provided by Infura is to reduce the amount of traffic if there is a rate limit response. In the Ethereum case, there are two services to choose from if one is being faulty. An attack would need to send an excessive amount of traffic to the services. If the services supplying the wallet are down for the default wait time for swap redemption, an initiator of a swap may be able to deny their counterparty from responding in time.

Mitigation

Alert the services if you expect that your application will create a large amount of requests.

Remediation

Stand up nodes and infrastructure to support the Atomex application specifically and do not rely on third parties that serve the community.

Status

The Atomex team has responded that they are working on their own infrastructure to address this issue and have not addressed the issue at the time of this verification.

Verification

Unresolved.

Issue H: Hash Time Locked Contract (HTLC) Preimage Secret is Not Stored Safely in Memory

Location

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Core/Swap.cs#L97

Synopsis

The preimage of a swap secret is kept in the client in unprotected memory. If a client is compromised before revealing the secret, this would allow a counterparty to be able to redeem a swap and cancel their end before the other party can reveal and claim their swap.

Impact

Loss of swap funds as one party is able to derive the pre-image before it is revealed.

Preconditions

The attacker must be able to compromise the victim's computer and read the insecure preimage bytes from memory.

Feasibility

This is not highly feasible as the preimage secret is only sensitive in the beginning of the swap and the attacker must compromise the target machine.

Technical Details

A crucial component of an atomic swap protocol is the commit reveal stage of a secret. This secret ensures that a counterparty is not able to redeem a locked amount of funds before the other party is able to secure their side of the swap. For example, if A initiates a swap, they will lock funds into a contract with the image of a secret. Party B will then not lock their side of the funds into a contract that requires revealing the preimage to the image that A created and attempt to compromise A. This area is the primary concern for the security of a swap. If B is able to deduce the image of the swap before the swap timeout that A created, B may then redeem A's lock without deploying B's side of the swap contract.

Mitigation

Use a method such as those recommended in Issue A to ensure that the preimage is also kept in a secure and encrypted state.

Status

The team responded with clarification that the client software will not be initiating atomic swaps, therefore it is unnecessary to protect the secret in memory as by the time the client receives it from the blockchain, the secret is meant to be public. This information makes this issue invalid.

Verification

Invalid Issue.

Suggestions

Suggestion 1: Use Certificate-Pinning for all Atomex API Endpoints

Location

https://github.com/atomex-me/atomex.client.core/tree/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Web

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Updates/Components/HttpMetadataProvider.cs

Synopsis

Atomex opens HTTPS connections to atomex.me. In order to fend off most attacks on the public-key infrastructure (PKI) that backs TLS, we recommend performing certificate authority (CA) pinning.

Mitigation

When establishing a connection to an API endpoint, perform a check that the used certificate was issued by Let's Encrypt, the CA that issued the atomex.me certificate. (Note that this needs to be updated if you intend to switch away from Let's Encrypt.) While this is not an immediate security issue, it may be used as a vector to interfere with the service.

Status

The Atomex team has responded that they will delay using certificate and CA pinning until they have switched away from using Let's Encrypt as a CA. However, discontinuing use of Let's Encrypt is not a prerequisite for pinning the CA and, as a result, our recommendation for certificate pinning of all API endpoints remains.

Verification

Unresolved.

Suggestion 2: Explore Paths to Reduce Single Points of Failure

Location

https://github.com/atomex-me/atomex.client.core/blob/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Subsystems/Terminal.cs

https://github.com/atomex-me/atomex.client.core/tree/2cf279bfd4202e90b9534b0f797b428e9c7e3d87/Atomex.Client.Core/Web

Synopsis

Currently, Atomex's use of central services and the Atomex API can be single points of failure. We encourage the Atomex team to explore ways to migrate these centralized components to decentralized technologies to remove these as single points of failure and reduce or remove maintenance and operational costs. However, for a fully decentralized system, the central relay would need to be a P2P network.

Mitigation

Research options to further decentralize Atomex by replacing centralized components with P2P technologies to reduce risks in single points of failure. A starting point could be the libp2p gossipsub protocol and to reduce the reliance on services like Etherscan or tzStats, the wallet could contain a light client functionality to track account balances. If and when an appropriate protocol has been identified, we recommend implementing it in Atomex to achieve a more decentralized exchange.

Status

The Atomex team stated they are considering the possibility of creating a decentralized order book and are currently in the research stage.

Verification

Resolved.

Suggestion 3: Increase Test Coverage

Location

https://github.com/atomex-me/atomex.client.core/tree/2cf279bfd4202e90b9534b0f797b428e9c7e3d87

Synopsis

While reviewing the tests, we found that much of the functionality of the core is not tested. For example, the swap manager contains no unit or integration tests. Many of the other critical functions like `PayAsync()` are also not tested.

Mitigation

Add more unit tests coverage for as much of the functionality as possible to ensure that the implementation operates as expected.

Status

The Atomex team issued a commit adding new tests to the code base. They have also responded that additional tests will be added in the future in an attempt to reach full coverage.

Verification

Partially Resolved.

Suggestion 4: Add Hardware Wallet Support

Location

https://github.com/atomex-me/atomex.client.core/tree/2cf279bfd4202e90b9534b0f797b428e9c7e3d87

Synopsis

Currently keys are being stored in LiteDB on the client's computer. If the client is compromised or inspected by a rootkit, then keys may be lost.

Mitigation

Generally, it is advisable to store large amounts of funds on a cold wallet that is on a secure hardware device external to the client computer. Implementing the ability to use hardware wallets like the Ledger device would add an extra layer of security for users.

Status

The Atomex team issued a commit making the transaction signing mechanism more transparent, in order to prepare for adding hardware wallet support. They have also responded that additional support is planned for the future. However, this feature has not been fully implemented at the time of this verification.

Verification

Partially Resolved.

Suggestion 5: Remove Unused Code

Location

https://github.com/LeastAuthority/atomex.client.core/blob/2f1bd7c5b62fe91abf7e144ce4c833ff3cb11ebd/Atomex.Client.Core/Blockchain/Bitcoin/QBitNinjaApi.cs

https://github.com/atomex-me/atomex.client.core/blob/master/Atomex.Client.Core/Wallet/Abstract/ICurrencyAccount.cs

https://github.com/atomex-me/atomex.client.core/blob/master/Atomex.Client.Core/Wallet/Abstract/IAddressResolver.cs

Synopsis

The locations provided show examples of files that have either been completely moved into a comment block or partially commented out and are ignored by the compiler as a result. It is unclear if unused code is not necessary, has a planned upgrade, or has issues that need to be addressed.

Mitigation

Remove all unnecessary files and comments that contain code that is not being used.

Status

The Atomex team has issued a commit, which removes the larger unused files. However, commented blocks of code that are not used are still present in the code base, which should be removed or incorporated as recommended.

Verification

Partially Resolved.

Suggestion 6: Provide a Method for Exporting Private Keys

Location

https://github.com/atomex-me/atomex.client.core/tree/2cf279bfd4202e90b9534b0f797b428e9c7e3d87

Synopsis

Currently the wallet application does not provide a clear way of extracting a private key or where the key is stored if it is needed.

Mitigation

Create a feature or provide documentation that supports a backup method and restoring of the private keys.

Status

The Atomex team has issued a commit adding private key export functionality to the client software, which includes messaging to the user about the security concerns of exporting a key.

Verification
Resolved.

Suggestion 7: Look for Memory Encryption Libraries for C#

Synopsis
To make attacks based on memory access more difficult, several tools that use cryptography have started to encrypt the parts of their memory that contain keys. This usually is not a perfect security measure, since very often the encryption key to that memory is also stored in memory. However, it does serve as a useful obfuscation technique.

Mitigation
Currently, it appears that a library for C# that provides this kind of functionality does not exist. However, we recommend to continue looking for the release of new libraries that support C#.

Status
The Atomex team has responded noting that the contents of a `SecureString` are stored encrypted when run on OS Windows. However, these protections are not available on other operating systems and, as a result, we recommend looking for the release of new libraries that support C#.

Verification
Partially Resolved.

Suggestion 8: Improve Documentation

Location
https://github.com/atomex-me/atomex.client.wpf/tree/cab4af61379d13adab90b65d526187083a799f91

https://github.com/atomex-me/atomex.client.core/tree/2cf279bfd4202e90b9534b0f797b428e9c7e3d87

Synopsis
We found there to be a lack of code comments and documentation which resulted in difficulty in building and testing the code.

The desktop client requires additional information on the proprietary dependencies needed in order to build the desktop client, such as Microsoft libraries. Furthermore, the core library requires information and documentation on the instructions to build, install, and test the code and which operating system or version of Windows is required.

Mitigation
We recommend including code comments and documentation that would allow new contributors and reviewers to understand the core library and desktop client more easily and efficiently, deduce the role of a function, and reduce some of the existing complexity required to build, develop and test the code.

Furthermore, the project would significantly benefit from additional documentation including:

- A comprehensive README file;
- General description of the implementations;
- How certain problems in the implementation are solved;
- The intention behind the design and expected behaviors;
- Description of the dependencies (e.g. Better-Call-Dev, Baking-Bad, Etherscan); and

- Certain security limitations as described in Issue A.

Status

The documentation has not been changed or improved at the time of verification. As a result, we recommend that the suggested improvements to the documentation be implemented.

Verification

Unresolved.

Recommendations

We recommend that the unresolved *Issues* and *Suggestions* stated above are addressed as soon as possible and followed up with verification by the auditing team.

We commend the Atomex team for their effort in designing in such a way that considers potential security implications. However, the errors identified in the cryptographic components of the system should be addressed in order to more effectively secure the use of the wallet. We suggest several areas of improvement, including better management of swaps data and relying less on third-party services or centralized API's in supporting the Atomex application.

We also recommend improved documentation coverage through the addition of code comments and more comprehensive test coverage, as well as better instructions on how to build, install, and test the code. The application of these development best practices will facilitate an easier and more efficient understanding for users, implementers and reviewers of the code, improving the overall security as a result.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our 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 protect 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, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet 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.