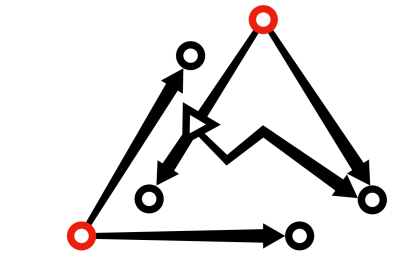

Least Authority

PRIVACY MATTERS

Umbra-js

Security Audit Report

ScopeLift

Final Report Version: 25 May 2021

Table of Contents

Overview

Background

ScopeLift requested that Least Authority perform a security audit of `umbra-js`, an off-chain JavaScript library that implements the basic cryptographic scheme for interacting with the Umbra Protocol and used for building Umbra-enabled Web3 applications in Node.js or in the browser.

Umbra is a protocol for enabling stealth payments on the Ethereum blockchain. It aims to enable privacy preserving transactions where the receiver's identity is only known to the sender and receiver.

Project Dates

- **May 3 - May 11**: Code review *(Completed)*
- **May 14**: Delivery of Initial Audit Report *(Completed)*
- **May 19 - 21:** Verification *(Completed)*
- **May 22:** Delivery of Final Audit Report *(Completed)*
- **May 25**: Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Jan Winkelmann, Cryptography Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of `umbra-js` 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:

- Umbra-js: https://github.com/ScopeLift/umbra-protocol/tree/master/umbra-js

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

`48991f0fabbcf19bb9882903c6ccf8e21a4f9c6d`

For the verification, we examined the Git revision:

`3796fa747cc568cb7e48f552675734a9cd564ee9`

For the review, this repository was cloned for use during the audit and for reference in this report:

https://github.com/LeastAuthority/Scopelift-Umbra-js

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

In addition, any dependency and third party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Umbra-js README.md: https://github.com/ScopeLift/umbra-protocol/blob/master/umbra-js/README.md
- Umbra Protocol `README.md`: https://github.com/ScopeLift/umbra-protocol/blob/master/README.md
- Umbra FAQ: https://app.umbra.cash/faq
- Umbra-js code comments: `umbra-js-docs.zip` shared with Least Authority via Slack on 3 May 2021

In addition, this audit report references the following documents:

- A. Antipa, D. Brown, A. Menezes, R. Struik, S. Vanstone, 2003, "Validation of Elliptic Curve Public Keys." In: Desmedt Y.G. (eds) Public Key Cryptography — PKC 2003. PKC 2003. Lecture Notes in Computer Science, vol 2567. Springer, Berlin, Heidelberg. [ABM+03]
- D. Brown, 2009, "Standards for Efficient Cryptography 1 (SEC 1)." 2009, Certicom Corp. [B09]
- E. Barker, A. Roginsky, 2019, "Transitioning the Use of Cryptographic Algorithms and Key Lengths." NIST Special Publication 800-131A Revision 2, 2019, [BR19]

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- General use of cryptography;
- Vulnerabilities within each component as well as secure interaction between the components;
- Key management implementation, including secure private key storage and proper management of encryption and signing keys;
- Attacks that impacts funds, such as draining or manipulating of funds;Protection against malicious attacks on the library;
- Denial of Service (DoS) attacks and other methods of exploitation;
- Adversarial actions and protection against malicious attacks on the library;
- Exposure of any critical information during interactions with users and external components;
- 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

The Umbra Protocol enables stealth payments on the Ethereum blockchain, allowing users to send and receive ETH, in addition to supported tokens, while preserving the privacy of the receiver. Each transaction is sent to a new address that is non-interactively generated by the sender, but is controlled by the owner of a publicly known address (the receiver). The Umbra Protocol stealth addresses are generated such that they can only be identified by the sender and receiver, assuming that both parties adhere to the rules of the protocol. While investigation and analysis of these assumptions of the Umbra Protocol were considered out of scope for the audit, the core focus of our security review was `umbra-js`, a JavaScript library that implements the cryptographic operations used for building applications on top of and implements core functionality of the Umbra Protocol.

System Design

We examined the `umbra-js` system design through review of the Umbra Protocol documentation (see Documentation) and the `umbra-js` implementation. We did not identify instances where the implementation deviates from the provided documentation. In addition, we did not identify issues with the

generation of stealth addresses that only the sender and receiver can identify, assuming that both sender and receiver are honest and, in particular, that the sender does not disclose the random number used to generate such a stealth address.

Random Number Generation

Through close examination of random number generation and the use of random numbers for computing stealth addresses, we identified a vulnerability that may allow an attacker to link a stealth address to a receiver. Specifically, stealth addresses provide only 64 bits of security by default, which is insufficient protection against brute force attacks. As a result, we recommend using the full 256 bits of the curve's scalar field element as the random number (Issue A).

Encryption Scheme

The Umbra Protocol, as implemented by `umbra-js`, makes use of a component for public-key encryption based on the ephemeral Diffie-Hellman protocol, similar to the Elliptic Curve Integrated Encryption Scheme (ECIES) [B09]. It differs in that the y-coordinate sign bit of public keys is not sent and XOR is used for encryption, limiting messages to 256 bits. We examined the use of XOR to encrypt the generated random number and did not identify any issues. We found that the requirements for secure encryption, specifically the uniqueness of key material, are fulfilled by construction through the use of ephemeral keys.

For the purpose of gas cost efficiency, the y-coordinate is not explicitly published on-chain, even in compressed form. Instead, it is inferred exclusively from the x-coordinate. This inference is ambiguous, particularly since there are often two valid points that have the same x-coordinate. While this presented some initial challenges in reasoning about the security of this approach, we verified the security of this approach in the context of the Umbra Protocol, as the two possible y-coordinates are one value and the negative of the value. Due to the affine formula for negations in short Weierstrass curves $-(x, y) = (x, -y)$, an arbitrary choice only influences the sign of the point. As a result, Diffie-Hellman computations will always result in the same shared secret (which is derived from the x-coordinate only), since $[s](-[r]G) = -[sr]G$ always holds.

Point Validity Checks

We examined the use of compressed and uncompressed public keys in `umbra-js` and found that uncompressed public keys are not validated, making the protocol vulnerable to invalid curve attacks. In order to avoid invalid curve attacks, validity needs to be explicitly checked for all uncompressed points. As a result, we recommend consistent checks for whether uncompressed public keys fulfill the curve equation by adding a check to the respective part in the constructor of the `KeyPair` class (Issue B).

JavaScript Execution Environment Limitations

The JavaScript execution environment used for running `umbra-js` fulfills the requirement for quick development of native UI applications and can be used in the browser. However, it is worth noting that this comes with security trade-offs, since a JavaScript environment is not optimal in meeting high security requirements. In particular, a JavaScript execution environment presents challenges in hiding secret data in memory and preventing the data from being written to disk during swapping. After swapping to disk, it may be difficult to reliably erase data, as Solid State Drives (SSDs) are known to make wiping difficult or impossible.

In addition, a JavaScript execution environment significantly limits the ability to perform constant-time cryptography. `umbra-js` utilizes the library `noble-secp256k1` for cryptographic operations and the `noble-secp256k1` documentation explicitly states this limitation in the JavaScript execution environment. However, we did not identify implementation-level security vulnerabilities in `umbra-js`. We acknowledge that these are known security trade-offs, however, we recommend providing users with

explicit documentation of the design choices and security trade-offs made in order to fulfill the system requirements (Suggestion 3).

Dependencies

We identified a concern in the use of a third party dependency. In particular, we found that `umbra-js` utilizes, `noble-secp256k1` version 1.2.0, which contains a slight bias in the private key generation. As a result, we recommend updating `noble-secp256k1` to the latest version (Issue C).

Code Quality + Documentation

The `umbra-js` code is well written and the repository is logically organized. Test coverage of `umbra-js` is generally sufficient, however, the code used for updating public keys on the Ethereum Name Service (ENS) and the Crypto Name Service (CNS) is not tested. The code comments in these particular files indicate that tests would require publishing data on-chain using Ganache, which has yet to be implemented for ENS and CNS. Until this is set up in the testing environment, we recommend to mock the provider, so the code can be tested for harmful behavioral regressions (Suggestion 1).

Documentation

The project documentation, including the `README.md` files and the Umbra FAQ, is helpful in explaining the intended functionality of the system and providing a detailed description of the Umbra Protocol and its security limitations. We also found the code comments to be sufficient in their coverage and accurate description of functions.

However, we identified several areas where the documentation would benefit from further improvement. We recommend providing a clear protocol specification (Suggestion 2) and clearly documenting the security trade-offs in the systems design (Suggestion 3). Improvements to the project documentation facilitate a better understanding of the system by contributors and reviewers, thus limiting the potential for missed issues. In addition, comprehensive user documentation provides transparency and encourages best practices for interacting with the system.

Scope

We found the scope of the audit to be sufficient.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, 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: Stealth Addresses Provide Only 64-Bit of Security by Default…	Resolved
Issue B: Group Membership of Public Keys Not Checked	Resolved
Issue C: noble-secp256k1 Contains Bias in Private Key Generation	Resolved
Suggestion 1: Test Domain Updating Code	Resolved
Suggestion 2: Provide a Clear Protocol Specification Document	Unresolved

Suggestion 3: Document Security Trade-Offs	Resolved

Issue A: Stealth Addresses Provide Only 64-Bit of Security by Default

Location

`src/classes/RandomNumber.ts#L37`

Synopsis

In the Umbra Protocol, as implemented by `umbra-js`, stealth addresses are derived from spending keys by scalar multiplication with random numbers. However, these numbers only contain 128 bits of randomness by default, in addition to 128 bits that can be used for short messages but are zeroes by default. In elliptic curve cryptography, 128 bits of randomness translates to 64 bits of security, which is insufficient.

Impact

Stealth addresses may be brute force derivable from spending keys and vice versa.

Preconditions

The attack assumes that the sender does not fill up the upper 16 bytes of the random number with cryptographically strong randomness on their own. In addition, the attacker needs access to the stealth address public key, which is given if the target account has signed a transaction (e.g. a withdrawal).

Feasibility

The attack only requires to run a standard brute force discrete log algorithm, such as the Pollard-Rho method, which is feasible for short secret values. For similar problems in secp256k1, practical attacks already exist, with readily available code. As a result, we estimate that it is not difficult to adapt the code to the context of the Umbra Protocol.

Technical Details

In the Umbra Protocol, spending keys as well as stealth addresses are points on the elliptic curve secp256k1, where secret keys have a size of 256 bits. If a spending key pk is given, any associated stealth address key pk_s is computed as the secp256k1 scalar multiplication of a random number r, supposed to be from the secp256k1 scalar field, with the spending key, specifically,

$$pk_s = [r]\,pk$$

For an attacker, associating any stealth address public key to a given spending key is as difficult as finding a discrete log relation between the two curve points. For cryptographically strong 256-bit random numbers `r`, this should provide roughly 128-bit security, assuming that the fastest algorithms, such as the Pollard-Rho method, have runtimes in the order of `2^(len(r)/2)`. The random number is then encrypted and published on-chain.

However, the Umbra Protocol only uses the lower 16 bytes of the random number, overwriting the upper 16 bytes with zeros by default, which then only gives roughly 64 bits of security. The purpose is to give the sender 16 bytes of space to transmit encrypted short messages, along with the actual randomness to the public key owners.

NIST 800-131A [BR19] presents the security level to target, which for all classes of protocols does not consider anything below 112-bit security and corresponding elliptic curve key lengths of 224 bits to be sufficiently secure.

Mitigation

If a message or memo field is required and a security level below 128 bits is acceptable, we recommend choosing the bit lengths of the memo field `f_memo` and random data `f_rand`, such that the sum is 256 bits. This results in a security level of at least `len(f_rand)/2`.

From these bit lengths, a uniformly random scalar field element (i.e. a number less than group order n) must be computed. To do so, we recommend using a hash function or an extractor like HKDF-Extract (which is just a way of using HMAC) to extract a 256-bit value `sk_stealth` from the SMS and random value. If `sk_stealth >= n`, pick a new random number and try again. This should be repeated until an appropriate value is found. Submit the encryption of `f_memo` and `f_rand` as usual and the receiver can try to decrypt, extract the value, and check if the stealth address matches. If it does not match, or if the announcement extracts to a value `>= n`, they can then deduce that they are not the recipient.

In addition, we recommend updating the Umbra documentation to reflect the changes to the protocol, as well as the reasoning behind the chosen lengths and security parameters.

Remediation

We recommend using the full 256 bits from the curve's scalar field as the random number.

In addition, we recommend updating the Umbra documentation to reflect the changes to the protocol, as well as the reasoning behind the chosen lengths and security parameters.

Status

The ScopeLift team has updated the code to use the entire 256 bits of randomness for the random number. Additionally, the update removes support for memos or messages, leading to full 128-bit security.

Verification

Resolved.

Issue B: Group Membership of Public Keys Not Checked

Location

`src/classes/KeyPair.ts#L59`

Synopsis

In `umbra-js`, parties may read public keys from untrusted sources. These public keys are binary representations of points that are supposedly on the elliptic curve. However, not all binary strings are representations of valid curve points.

Impact

Publishing results of computations on invalid curve points may leak sensitive data that contains or allows inference or linking of key material. This issue specifically impacts the handling of uncompressed curve points, since compressed curve points are implicitly checked for validity during decompression. In addition, not checking for public key validity puts the privacy of the recipient of a transaction at risk.

Preconditions

The public key must enter the system in uncompressed form and not be validated. The attacker must pick a point in a compatible, yet weak curve.

Feasibility

The attack requires some computational resources, but not enough to make the attack infeasible.

Technical Details

Elliptic curve public keys are points and often are described using the two affine coordinates x and y. However, only the points that satisfy the respective curve equation are valid public keys. Public keys entering the system need to be validated. For compressed points this happens implicitly, because the y coordinate is not transmitted in full and is found by solving the curve equation, which means the equation is trivially satisfied or the decompression fails. For uncompressed points, however, validity needs to be checked explicitly.

Failure to check public key validity could make the protocol vulnerable to invalid curve attacks described in [ABM+03], which further lead to small subgroup attacks and the possibility of weak shared secrets being used, allowing an attacker to compute encryption keys or stealth addresses.

Remediation

We recommend consistent checking of whether uncompressed public keys fulfill the curve equation by adding a check to the respective part in the constructor of the `KeyPair` class.

Status

The ScopeLift team updated the `umbra-js` library to check if the elliptic curve equation is satisfied when working with uncompressed keys.

Verification

Resolved.

Issue C: noble-secp256k1 Contains Bias in Private Key Generation

Location

`noble-secp256k1/`

Synopsis

`Umbra-js` utilizes `noble-secp256k1` version 1.2.0 which contains a slight bias in the private key generation.

Impact

The bias slightly increased the probability of generating insecure encryption and stealth keys, which would have led to linkability of stealth addresses to real addresses.

Remediation

We recommend updating `noble-secp256k1` to the latest version.

Status

The ScopeLift team updated `noble-secp256k1` to version 1.2.5, as recommended.

Verification

Resolved.

Suggestions

Suggestion 1: Test Domain Updating Code

Location

umbra-js/test/DomainService.test.ts#L57-L59

umbra-js/test/cns.test.ts#L68-L71

umbra-js/test/ens.test.ts#L76-L79

Synopsis

The functions for updating public keys on ENS and CNS are currently not tested. The code comments in these particular files indicate that tests would require publishing data on-chain using Ganache, which has yet to be implemented for ENS and CNS.

Testing helps to identify harmful behavioral regressions. Furthermore, a robust test suite improves the code quality by providing a mechanism through which new developers and reviewers gain an understanding of how the code works through use cases. In doing so, tests provide visibility into whether the code behaves as expected and help to identify the potential for errors, bugs, and edge cases.

Mitigation

We recommend immediately testing the behavior using Ganache, as planned, or mocking the provider as an interim solution so that the requirement for sufficient test coverage is fulfilled.

Status

The ScopeLift team updated the `umbra-js` repository to now contain tests for setting public keys on ENS and CNS addresses, as well as the `DomainService` class that abstracts over the operations of ENS and CNS addresses. As a result, there are no remaining under-tested parts of the code.

We found that some tests have a tendency to time out, which could be resolved by increasing the timeout limit. We recommend taking this extra measure to reduce the rate of false positive test failures.

Verification

Resolved.

Suggestion 2: Provide a Clear Protocol Specification Document

Synopsis

A protocol specification clearly lists the goals and assumptions of a protocol, in addition to the mechanisms used to achieve the goals and assumptions. The Umbra Protocol is currently missing a protocol specification. While the Umbra FAQ provides meaningful and helpful information (e.g. "How does it work?"). Several important details, such as the hashing of the shared secret and that the random number is an element of the secp256k1 scalar field, are missing.

A protocol specification (i.e. a self-contained reference) benefits security in multiple ways. First, it allows reasoning about the protocol in all its details at an abstract level, without being concerned with implementation details. Second, the act of precisely writing down the assumptions, goals, and mechanisms provides a better understanding of the interactions between system components. For example, a security issue (e.g. Issue A) may have been caught during such an undertaking. Finally, a

protocol specification allows auditors to verify that the implementation matches the protocol specification, instead of having to reverse-engineer the specification from the implementation.

Mitigation

We recommend providing a specification of Umbra Protocol. We suggest referencing the structure of IETF RFCs as a helpful guide.

Status

The ScopeLift team has responded that they intend to write a formal specification or EIP. However, due to the level of effort required for such an undertaking, the suggestion has not been implemented at the time of this verification.

Verification

Unresolved.

Suggestion 3: Document Security Trade-Offs

Location

Umbra FAQ

Synopsis

The JavaScript execution environment used for running `umbra-js` fulfills the requirement for quick development of native UI applications and can be used in the browser. However, a JavaScript environment is not optimal in meeting security requirements as it presents challenges in hiding secret data in memory and preventing the data from being written to disk during swapping. In addition, a JavaScript execution environment significantly limits the ability to perform constant-time cryptography. `umbra-js` utilizes the library `noble-secp256k1` for cryptographic operations and the `noble-secp256k1` documentation explicitly states this limitation in the JavaScript execution environment.

We did not identify implementation-level security vulnerabilities in `umbra-js`, resulting from this limitation in the Umbra Protocol. However, it is important to notify users of these trade-offs in order to promote transparency and to encourage safe usage of the system.

Mitigation

We acknowledge that these are known security trade-offs, however, we recommend providing users with explicit documentation of the design choices and security trade-offs made in order to fulfill the system requirements.

Status

The ScopeLift team has updated the FAQ, which now elaborates further on the trade-offs made in `umbra-js` and sufficiently informs users about the design decisions as it relates to security.

Verification

Resolved.

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.