Fluidkey Stealth Account Kit

Smart Contract Security Assessment

May 24, 2024

ABSTRACT

Dedaub was commissioned to perform a security audit of the Fluidkey Stealth Account Kit, an open source TypeScript library containing the core cryptographic functions used by Fluidkey. The Fluidkey system allows users to receive funds in private “stealth” addresses, that are generated for each transfer without interaction with the recipient. The library enables anyone to independently generate stealth smart accounts owned by them, and recover any related funds.

SETTING & CAVEATS

This audit report mainly covers the TypeScript code in the public repository fluidkey/fluidkey-stealth-account-kit at commit `518a4dec1ccdb3a1d61dba62b6cf8e124a89367d`.

Two auditors worked on the codebase for 2.5 days on the following files:

```
src/
├── extractViewingPrivateKeyNode.ts
├── generateEphemeralPrivateKey.ts
├── generateKeysFromSignature.ts
├── generateStealthAddresses.ts
├── index.ts
├── predictStealthSafeAddress.ts
└── predictStealthSafeAddressTypes.ts
```

As part of the audit, we also reviewed the fixes for the issues included in the report (delivered at commit `192a2260c0a254d28951519f1bbef1f6f4e44312` in the same repository) and we found that they had been implemented correctly.

The audit covers the open source library, and not the Fluidkey system itself. Its use in the overall system was only inferred from the documentation provided, the security of the system obviously depends on the proper application of the library. The audit mainly focuses on the use of the library by an end user who wishes to independently recover their stealth accounts, assuming the overall system behaves as expected. Some design aspects are also discussed in the report (see PROTOCOL-LEVEL CONSIDERATIONS below).

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

PROTOCOL-LEVEL CONSIDERATIONS

The Stealth Account Kit is a Typescript library that can be used either by end-users directly, or integrated in the Fluidkey app. In this section we discuss some aspects of its current integration within the Fuildkey app, and in particular its comparison with ERC-5564, which provides both a scheme-agnostic specification for stealth addresses as well as a specific implementation scheme. Note that these issues concern the library's application, rather than the library itself, and there are plans to address them in future versions of the Fluidkey app.

ID	Description	STATUS

P1	Private spending key accessible by application code	INFO
In the Fluidkey design, both spending and viewing private keys are generated from a message signature, created by the user's wallet and provided to the application code running in the browser. The spending key is then discarded and not sent to the server; it is only used on the client side to recover the private key of stealth addresses. Although the legitimate Fluidkey application code does not store or transmit the spending private key, if the Fluidkey server gets compromised it could potentially serve malicious javascript code to the user. That malicious code would have access to the private key, hence the overall security of stealth addresses currently relies on the security of the Fluidkey server. A potential way to address this issue in future versions of the app is to give the possibility to senders to generate stealth addresses directly, without interacting with the Fluidkey server, similarly to the scheme described in ERC-5564.		

P2	The Private viewing key is needed to generate stealth addresses	INFO
In the current Fluidkey app, the private viewing key is used by the server to derive the ephemeral keys needed to generate stealth addresses. The user can reconstruct the ephemeral keys locally to obtain the private key for each stealth address, avoiding the need for announcements. This is not only different from the specific scheme described in ERC-5564, but it also deviates to some extent from the scheme-agnostic specification, which generally expects any scheme to allow generating stealth addresses using only the recipient's public keys. A potential way to address this issue in future versions of the app is to allow the sender to generate addresses using a randomly generated ephemeral key, similarly to the		

scheme described in ERC-5564. The stealth account kit library could be easily applied to this task, by simply replacing the `generateEphemeralPrivateKey` function.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists.

	• Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed” or “acknowledged” but no action taken, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

ID	Description	STATUS
M1	The exact signed message is unspecified	**RESOLVED**

In the Fluidkey system, the spending private key is generated from a signed message via `generateKeysFromSignature` and is not stored anywhere. Then, the same message needs to be signed again in the future every time the spending private key is used. As a consequence it is vital that the user can produce this message by himself, without any interaction with the Fluidkey server, in order to be able to recover their funds in case of an emergency.

However, the exact message to be signed is not generated by the library, `generateKeysFromSignature` simply receives the signature as an argument. Moreover, the example given in `example.ts` shows that the signed message actually contains a secret, which is obtained from a PIN chosen by the user. But neither the example nor the documentation provide instructions for generating this secret, hence the user would be unable to recover his funds without interacting with the Fluidkey server to obtain the exact message to sign.

As a consequence, we recommend adding clear documentation about the format and content of the signed message, as well as code that produces the complete signature needed to generate the private keys.

LOW SEVERITY:

ID	Description	STATUS
L1	Missing code to generate the private spending key	**RESOLVED**

To use the received funds, the user needs to recover the private key of each stealth address. This is done by recovering the ephemeral key, generating the shared secret, and computing `spendingPrivateKey * h(sharedSecret)` (mod group order).

However, there is currently no code and no test performing this computation. As a consequence, recovering the key in case of an emergency, without interacting with Fluidkey servers, would require unrealistic expertise from the user.

Hence, similarly to M1, we recommend adding code that fully performs all necessary computations and is easy to use by end users.

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Testing limited to simple unit tests	**RESOLVED**

The library contains simple unit test cases that cover independent invocations of each function. However, there is no testing of the interaction of these functions. We recommend adding end-to-end test cases for the complete flow of actions in the system, ensuring that stealth addresses can be successfully generated by the server and then recovered by the user.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Security Suite.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.