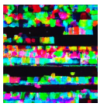


ZKSECURITY

Audit of Silent Protocol Smart Contracts

July 7th, 2023

Introduction

On June 26th, 2023, zkSecurity was commissioned to perform a security audit of Silent Protocol's Ethereum smart contracts. For the next two weeks, one consultant reviewed the solidity smart contracts in search of bugs.

A number of observations and findings have been reported to the Silent Protocol team. The findings are detailed in the latter section of this report.

Silent Protocol's smart contracts were found to be of high quality, accompanied with thorough documentation, specifications, and tests.

Note that security audits are a valuable tool for identifying and mitigating security risks, but they are not a guarantee of perfect security. Security is a continuous process, and organizations should always be working to improve their security posture.

Scope

A consultant from zkSecurity spent two weeks auditing the Solidity smart contracts for the Silent multi-asset shielded pool (SMASP) application. The audit included all of the application's smart contracts, including the Silent token, the main SMASP logic, the compliance logic, an implementation of the Baby Jubjub curve, and a Groth16 verifier.

More explanation on the protocol and on the organization of the smart contracts can be found later in this report.

The audit was limited to the smart contracts themselves, and did not include the circuits encoding the protocol statements, or core dependencies like snarkjs.

Methodology

zkSecurity followed a number of methodologies and heuristics to audit the smart contracts. In this section we discuss some of them.

Replay attacks. The original Zether: Towards Privacy in a Smart Contract World white paper (Zether) talks about replay attacks and introduce nonces or epochs as a way to avoid these issues. While Silent Protocol uses none of these solutions, they are not vulnerable to replay attacks due to upgrading accounts instantly (on every user transaction). As such, they are not limited to one transaction per epoch (as in the original Anonymous Zether scheme) but could run into lock contention (multiple transactions trying to update the same accounts at the same time). As the application is used through "relayers", who are agents in charge of protecting the anonymity of the users and might know more about pending updates, the risk of lock contention could be reduced. Even then, on-chain colliding updates merely reveal that some addresses might be used by multiple people instead of one, while costing some gas to the relayers instead of the users.

From the Zether paper:

> *"Front-running. The very first problem with the simplistic version of Zether is that the ZK-proofs are generated w.r.t. a certain state of the contract. For example, the ZK-proof in a transfer transaction needs to show that the remaining balance is positive. A user Alice generates this proof w.r.t. to her current account balance, stored in an encrypted form on the contract. However, if another user Bob transfers some ZTH to Alice, and Bob's transaction gets processed first, then Alice's transaction will be rejected because the proof will not be valid anymore. Note that Bob may be a totally benign user yet Alice loses the fees she paid to process her transaction. We refer to this situation as the front-running problem. Burn transactions have a similar problem too: a proof that a ciphertext encrypts a certain value becomes invalid if the ciphertext changes."*

Rogue-key attacks. The Many-out-of-Many Proofs and Applications to Anonymous Zether white paper refers to a rogue-key attack allowing an attacker to deanonymize accounts by registering invalid public keys. Silent Protocol is not vulnerable to such attacks as they encode knowledge of the private key associated with the public key in the required registration proofs.

Malleability. Groth16 proofs are malleable, in the sense that one who observes a proof can trivially create a different proof for the same statement (without knowledge of the witness). This is not an issue for the Silent Protocol smart contracts as proofs (or derived identifiers) are never stored on-chain, and can never be replayed anyway. The replay is prevented due to proofs having the instant effect of mutating the state of the smart contract on verification, invalidating their own predicates.

Cofactor issues. The elliptic curve used for user accounts and compliance committee accounts is the Baby Jubjub curve, which is a twisted Edwards curve and for this reason does not have a prime order field (its order has a small cofactor of 8). This has been known to cause issues and is often referred to as "contributory behavior" (https://moderncrypto.org/mail-archive/curves/2017/000896.html, https://vnhacker.blogspot.com/2016/08/the-internet-of-broken-protocols.html, https://research.kudelskisecurity.com/2017/04/25/should-ecdh-keys-be-validated/). Attacks are usually about using a point in the wrong subgroup to force the result of a scalar multiplication to be predictable. We could find no instance where this was an issue.

Layer Zero Integration. The Silent Token is built on top of LayerZero's Omnichain Fungible Token template, which provides an integration checklist to follow for audits. As the integration is quite straightforward nothing stood out as an issue.

Recommendations

The following recommendations are based on the findings of the audit. We recommend that the Silent Protocol team take the following steps to improve the security and robustness of the protocol:

Fix findings. A number of findings have been reported in the latter section of this report. We recommend fixing all issues deemed important by the Silent Protocol team.

Review observations and discussions section. A number of design decisions were discussed in the observations and discussions section. We recommend documenting the rationale behind important decisions like the Groth16 parameters and how they were chosen/obtained, as well as the size of the anonymity set and how the anonymity set building algorithm was chosen.

Audit protocol circuits. We highly recommend auditing the Circom circuits related to the proofs verified by the SMASP smart contracts.

Observations and Discussions

During our review, we identified a number of concerns that were not necessarily security issues. We have included this section to discuss these as observations, in place of findings in this report.

Parameters for proofs

Silent Protocol currently uses snarkjs with the Groth16 backend to create and verify proofs.

The use of Groth16 as the zero-knowledge proof system implies that a trusted setup must be conducted safely. If not done correctly, this could imply leakage of so-called "toxic waste" which would allow anyone with access to it to forge invalid proofs.

During the engagement, the consultant had access to parameters output by the phase 1 of a trusted setup uploaded on S3. The phase 2 was automated via the following line:

```
$ echo "randomText" | snarkjs zkey contribute ${outDir}/${circuit}_0000.zkey
${outDir}/${circuit}.zkey --name="1st Contributor Name" -v
```

After discussions with Silent Protocol, it seems like the project intends on performing a proper ceremony. While the details of the ceremony are not yet known, we recommend reusing (if possible) well-known parameters and well-known methods.

The paper **Powers-of-Tau to the People: Decentralizing Setup Ceremonies** goes through a number of possible options:

> ***"The perpetual "powers-of-tau" ceremony was first run in a continuous mode, where contributions are still being accepted, by the team of the Semaphore project, a privacy preserving technology for anonymous signaling on Ethereum. The setup uses a BN254 elliptic curve and has had 71 participants so far. Other prominent projects later used this setup to run their own ceremonies on top, including Tornado.Cash Cas20 , Hermez network Her20, and Loopring Dev19."***

For example, the *perpetual powers of tau* available at **https://github.com/privacy-scaling-explorations/perpetualpowersoftau** seems to be one of the most promising options for phase 1 parameters.

For phase 2 of the trusted setup, one could study and reuse tools from other projects. For example, Namada (**https://namada.net/trusted-setup**), Tornado Cash (**https://tornado-cash.medium.com/tornado-cash-trusted-setup-ceremony-b846e1e00be1**), Telepathy (**https://docs.telepathy.xyz/protocol/circuits#trusted-setup-ceremony**), Ethereum's ceremony (**https://ceremony.ethereum.org**), Hermez (**https://docs.hermez.io/Hermez_1.0/about/security/#multi-party-computation-for-the-trusted-setup**), etc.

Anonymity set creation and size

The anonymity set size is a key metric for privacy protocols. The larger the anonymity set, the more difficult it is to link a transaction to a specific user.

Silent Protocol follows the approach of Zether and Monero, where users form their own anonymity set by mixing the paying and receiving accounts with unrelated others. Via the use of zero-knowledge proofs, other accounts are randomized while their balances remain untouched.

Currently, the encryption layer fixes this number at 8. That is, a transfer always modifies 8 accounts, of which 2 are the sender and receiver.

As was documented in **An Empirical Analysis of Traceability in the Monero Blockchain**, some client-side algorithms to form anonymity sets are trivial to deanonymize, and it can be tricky to tune them to be secure.

A good practice would be to document the rationale behind the choice of the anonymity set size and the algorithm used to form it.

For subscribed users, different functions are used (e.g. `` `_withdrawZeroFee` ``). Thus, a subscribed user must be careful not to form an anonymity set with accounts that do not have a subscription (otherwise it would be trivial to guess that the only subscribed accounts are the ones actually involved in the transaction).

In addition, it is notable that relayers get more information than on-chain observers. Relayers can see requests coming from certain IPs, for example. Page 44 of the Zether paper touches on that:

> ***"The idea of paying fees in a non-native currency is called economic abstraction and has been discussed intensively But15, Rub18. The concept is particularly interesting with respect to Zether as it is would make Zether a) more usable and b) more private. The major obstacle to this approach is that miners would need to mine these special 0 gas price transactions and properly rake the fee pool. A similar approach that circumvents the miner adoption problem is to have special delegator nodes that issue the transactions to the network. Users would send their transactions to delegators who will forward them to the miners and will pay the Ethereum gas fees. These delegator nodes could be rewarded in Zether by adding their Zether address to the transaction. The fee amount would not go to the fee pool but to that address. A general problem for any anonymous blockchain transaction is that transactions need to be relayed to either the miners or the delegators without revealing the original sender's identity. This problem can be alleviated through anonymous communication networks like Tor Tor."***

Thus, as a global recommendation, in addition to documenting the rationale behind the choice of the anonymity set size and the algorithm used to form it, we recommend giving good guidance to the user on how to use the protocol in a way that maximizes their privacy.

Users are prevented from self-exiting the application

While the soundness of the application (the impossibility for someone else to mess with their balance) is guaranteed by the use of zero-knowledge proofs (to the extent that the circuits are free of bugs), the liveness of the application (the ability for a user to withdraw their funds) is not.

The fact that the `` `withdraw()` `` function of the `` `SMASP.sol` `` smart contract is `` `onlyApprovedCallers` `` means that only the relayers can call it. Furthermore, the owner of the smart contract could also remove all approved callers if they wanted to.

Thus, the liveness of the application from the user's perspective relies on many assumptions. We recommend either removing the `` `onlyApprovedCallers` `` modifier, or documenting the rationale behind it.

Sanctioned users can still withdraw funds

Two compliance-related smart contracts are in charge of hosting lists of sanctioned Ethereum addresses as well as Silent Core addresses, in order to prevent them from using Silent Core. However, the `withdraw()` function (and related `withdrawInSilent` and `withdrawZeroFee` functions) of the `SMASP.sol` smart contract are not checking if the user withdrawing is in the sanction lists.

This means that the SMASP smart contract is currently not in charge of "freezing" sanctioned addresses. It simply prevents them from interacting with the protocol, and forces them to recover their funds and exit the application. This is a design choice that perhaps is not ideal, depending on the intention of the protocol.

Background on Silent Protocol and EZEE

This section provides an overview of the Silent Protocol smart contracts and the EZEE protocol.

Overview of the protocol

Silent Protocol is an anonymous Zether-based protocol deployed on Ethereum and aiming to provide privacy to its users. Users register their own accounts as keypairs on the Baby Jubjub curve (https://eips.ethereum.org/EIPS/eip-2494).

While deposits and withdrawals do not mask one side of the exchange and the amount being transferred, all other types of transfers within the application are encrypted and reveal no more than an anonymity set (of size 8). This part is called the encryption layer of the protocol.

In addition, the protocol uses Groth16 proofs to enforce the correct mutation of accounts in the anonymity set during movements of funds. Importantly, it enforces that the sender's account has enough balance, that the funds removed from their account are equal to the funds added to the recipient's account, and that all other accounts' balances are untouched and correctly randomized.

In order to increase the anonymity of the users, agents called relayers are the only ones allowed to submit transactions to the smart contract. Users thus pay relayers to submit their transactions, in exchange for obtaining anonymity on the Ethereum blockchain.

This collaboration between users and relayers does not stop at transfer of funds within the smart contract itself. Users can withdraw to special addresses and use whitelisted applications through the continued help of relayers as well.

A subscription is available to the users, in order to remove all fees encoded in the smart contract logic. Fees are payable in different cryptocurrencies, including the Silent token.

In order to remain compliant with anti-money laundering laws, the encryption layer of Silent Protocol forces users to encrypt all transactions to a compliance entity.

Later, this entity can look back at the history and arbitrary decrypt transactions. To add security to this process, the compliance entity is *decentralized* into a compliance committee.

The creation of the committee is done using a distributed key generation, so that at no point in time does the private key exist in one place. Committee members must then collaborate if they want to decrypt a transaction.

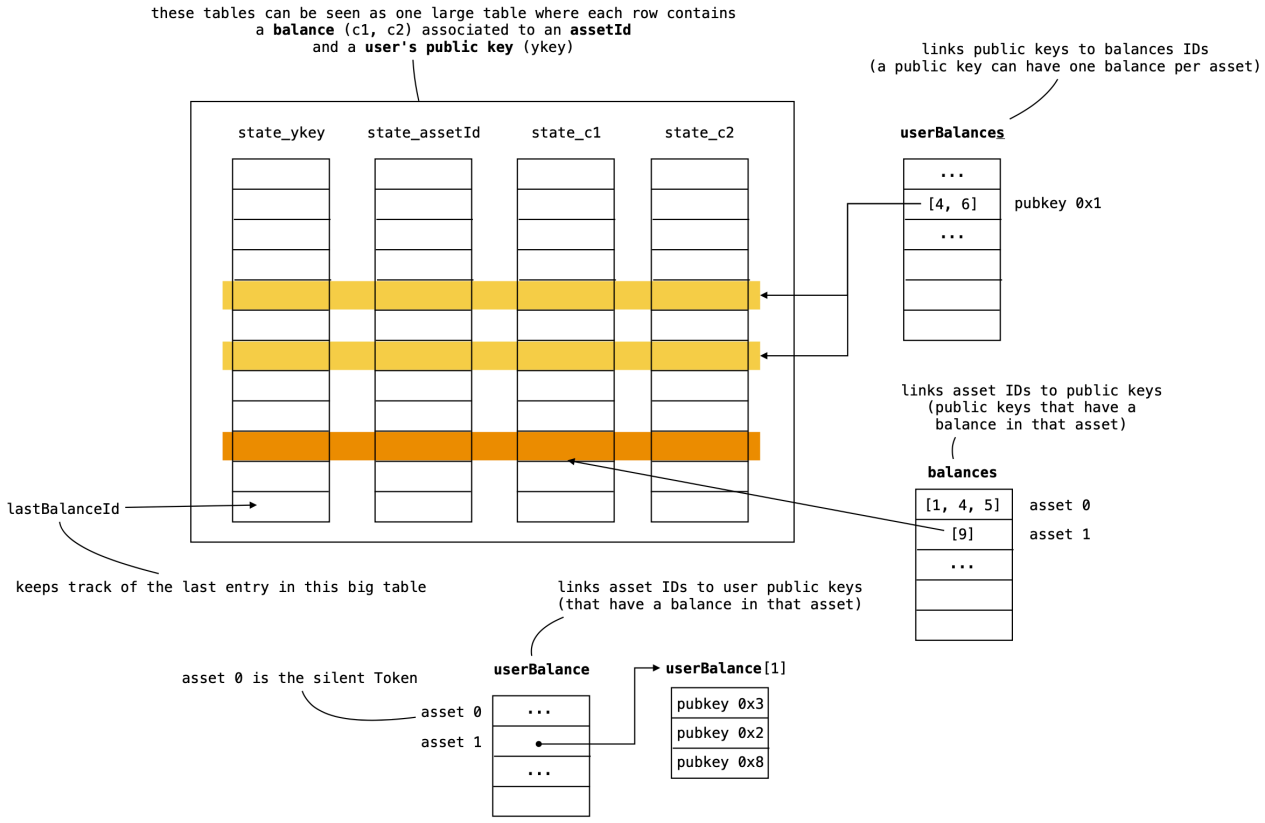
In addition, the Silent Protocol keeps track of sanctioned addresses (both Ethereum and Silent addresses) and prevents them from interacting with the protocol.

Smart Contracts Organization

In this section, we give a quick overview of the layout of the SMASP smart contracts.

The main smart contract is `SMASP.sol`, built on top of two OpenZeppelin's templates (Ownable2Step, ReentrancyGuard) and containing the "encryption layer" of Silent Protocol.

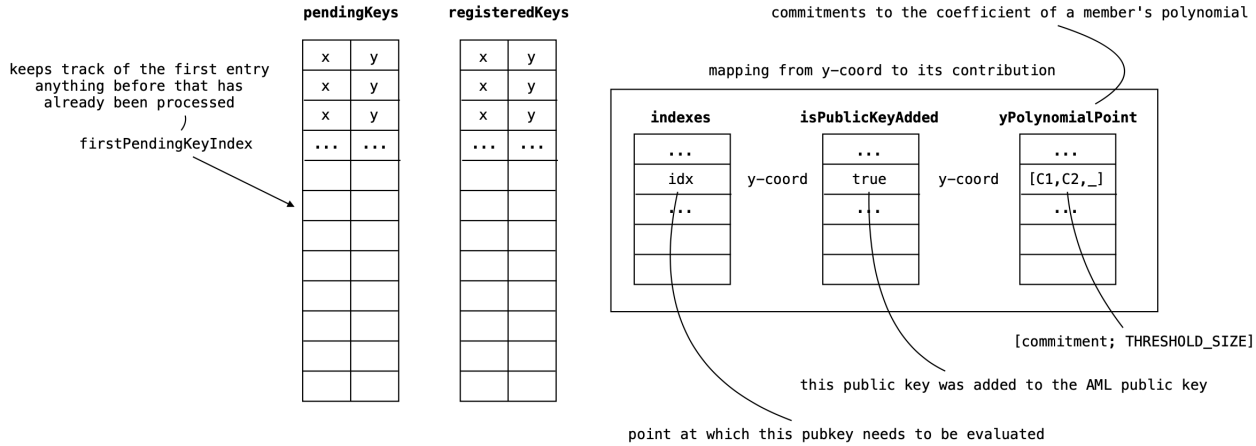
Within `SMASP.sol`, a number of tables are used to maintain the user states:



The state is also initiated with a number of smart contract addresses that are later used to externalize different aspects of the SMASP logic:

- silentTokenAddress: used as the ERC-20 smart contract for the Silent Token. It is also integrated into the LayerZero framework in order to facilitate cross-chain exchanges.
- sVault: used to store the fees collected on users using the SMASP application.
- verifyingKeys_1 and verifyingKeys_3: used to store the verifier keys used to verify Groth16 proofs.
- compliance: used for the compliance side of the application.
- uniswapRouter: used to calculate how much fees users should pay when they decide to pay fees in Silent Token.
- sanctionsListContract: used to maintain a list of sanctioned Ethereum addresses.
- silentSanctionsList: another list of sanctioned users, containing both a list of Ethereum addresses and of SMASP accounts.
- wethAddress: used with the uniswap router to compute fees. This is useful to create a path between an arbitrary token and the Silent Token, as there's most likely always be an exchange between ETH and approved ERC-20 tokens.

The compliance smart contract enforces the correct creation of a compliance committee, in which committee members take turns to contribute to a distributed key generation (DKG) process. This is done through the use of different tables as well:



Note that we had no access to the actual circuits during the audit. We assumed that circuits were correctly encoding the statements of the Silent Protocol white paper. Actual users would benefit from having a way to check that (open sourced) circuits correctly compile down to the verifier keys used on chain (in addition to obtaining assurance on the statements encoded in those circuits).

zkSecurity had access to the compiled version of the circuits. The verification of proofs was done by manually combining a Groth16 verifier smart contract (`ZKP.sol`, taken from the Hermez project) and smart contracts containing the verifier keys (obtained from the compiled circuits). In addition, `verifier.sol` exposed a usable interface for the `SMASP.sol` and `Compliance.sol` contracts.

A solidity implementation of the Baby Jubjub curve was also present, taken from https://github.com/yondonfu/sol-baby-jubjub/.

Findings

Below are listed the findings found during the engagement.

ID	COMPONENT	NAME
00	SilentToken.sol	Debug minting function is still present
01	SMASP.sol	Users can get lifetime subscription for a small one-time fee
02	Compliance.sol	Bricking the AML public key, and y-coordinates shenanigans
04	Compliance.sol	Lack of clear state transitions in the compliance contract
05	Compliance.sol	Committee members might be able to contribute several times
06	Compliance.sol	Index might not be unique in compliance setup
07	*	Solidity version is outdated
09	SMASP.sol	Uniswap contract version is outdated
0b	Compliance.sol	Remove function lacks explicit preconditions

00 - Debug minting function is still present

SilentToken.sol

Description. Assuming that zkSecurity is looking at the production-ready smart contracts, the following function that was meant for debugging only was still present and publicly callable by anyone:

```
/**
 * @notice HERE FOR TESTING PURPOSES ONLY.
 * TODO: Remove this function before production deployment
*/
function mint(address _recipient, uint256 _amount) public {
    _mint(_recipient, _amount);
}
```

This allowed any Ethereum user to call it and mint any amount of tokens to themselves.

Recommendation. In general, mixing development and production code is not ideal, as one can easily forget to remove test-only code.

01 - Users can get lifetime subscription for a small one-time fee

SMASP.sol

Description. Users of Silent Protocol can subscribe to the smart contract in order to reduce their fees. (Specifically, fees taken from the user when withdrawing.) The smart contract function to do that is `registerZeroFee`:

```
function registerZeroFee(
    FeeRegistrationInput calldata _input,
    uint256[8] calldata _proof
) external payable
```

One of the function argument, `_input.validUntil`, dictates until when (i.e. what block number) the subscription will be active for. This value is then encrypted and checked (via the use of proofs) when the user withdraws.

There are no checks on the discussed user-controlled value in the smart contract logic. This allows users to enter arbitrary values for `_input.validUntil`, for example, to get a lifetime subscription in exchange for a small one-time fee.

Note that there is a `subscriptionDuration` value stored in the state of the smart contract, but it's currently unused.

Recommendation. Use `subscriptionDuration` to ensure that the `validUntil` value set by the user is valid. In addition, add a negative test to test that a user can't subscribe for too long.

Client Response. Silent Protocol's monorepo did not contain the issue, only the contracts that were shared with zkSecurity did.

02 - Bricking the AML public key, and y-coordinates shenanigans

Compliance.sol

Description. Silent Core uses the Baby Jubjub curve for user keypairs, as well as the compliance committee keypairs. The Baby Jubjub curve, defined in **EIP 2494**, is an elliptic curve that is native to the circuit field of Ethereum's **BN254** pairing curve.

Baby Jubjub is a twisted Edwards curve, which means that the equation of the curve is of the form $\mathrm{ax}^2 + \mathrm{y}^2 = 1 + \mathrm{dx}^2\mathrm{y}^2$ for some specified a and d, and over a specified field $\mathbb{F}_\mathrm{r}$.

One particularity of such curves, is that for a fixed y-coordinate, one can recover two valid points (x, y) and $(-\mathrm{x}, \mathrm{y})$ on the curve.

For this reason, the y coordinate alone is not enough to recover a known point, and the logic often carries additional information (the sign of the x coordinate).

In this finding we explain how this ambiguity can be exploited to brick the AML public key in the `Compliance.sol` smart contract, and how it can be fixed.

How the Silent Protocol handles Baby Jubjub points. There are different ways that the smart contracts of the Silent Protocol handles these points:

- `SMASP.sol` stores them using the packed representation, but often handles them with the sign of the x coordinate separately. Specifically, the user always provides the sign of x. As far as we've looked, it seems that since everything stored in the smart contract is in packed representation, no issues arise from this ambiguity.
- `SilentSanctionsList.sol` stores Baby Jubjub points as y coordinates without the sign of x. This means that for a given Silent public key (x, y), the related public key $(-\mathrm{x}, \mathrm{y})$ will also be sanctioned. Which seems to be fine as the two addresses are related (if you know the private key of one, you can trivially derive the private key of the other).
- `Compliance.sol` mixes both storage in uncompressed format (i.e. (x, y)) as well as in y-coordinate.

The latter case is problematic as it leads to ambiguous scenarios. We want to avoid situations where a proof for one key can be reused for another, or a replay for the same key due to using a related key, or a wrong computation is accepted due to using a related key, etc.

One example that seems to be exploitable is in the compliance smart contract. When a committee member joins the committee, they register their public key with the `register()` function. This function stores the uncompressed public key ((x, y)).

Later, the member adds their contribution to the AML public key by calling `shareSecret()`. At this point, they can call it with the related key ($(-\mathrm{x}, \mathrm{y})$) but with a proof that uses the original key ((x, y)).

Note that anyone can observe a valid use of `shareSecret()` by a committee member, and attempt to front-run the transaction using the same proof but with an invalid point.

The previous malicious transaction will successfully execute (as demonstrated by the reproduction steps below), and the wrong elliptic curve point will be added to the AML key. Meaning that the secret share (along with its

distributed shares encrypted to other members) will be uncorrelated with the AML public key, thus transitioning the compliance contract into an unrecoverable state if that were to happen.

See the code of `shareSecret()` commented below:

```
function shareSecret(
    SecretSharingInput calldata _input,
    uint256[8] memory _proof,
    uint256[2] memory publicKey
) external {
    // TRUNCATED...

    // check if the proof is valid (this can use the original key)
    require(
        Verifier.verifySecretSharing(
            pkIndexes,
            _input,
            _proof,
            verifyingKeys_2
        ),
        "Invalid proof"
    );

    // TRUNCATED...

    if (!isPublicKeyAdded[yPublicKey]) {
        // this will add the related key to the AML public key
        updatePublicKeyJoin(publicKey);
        isPublicKeyAdded[yPublicKey] = true;
    }
```

Reproduction steps. You can add the following test to `test/compliance.test.ts` to reproduce the issue:

```
it("should reject if (-x, y) used instead of (x, y)", async function () {
    const { compliance, crypto } = await loadFixture(setup);

    // register a user
    let pks = [];
    const user = await crypto.genKeyPair();
    pks.push({
        xPublicKey: user.publicKey[0],
        yPublicKey: user.publicKey[1],
        index: bigint2array(genRandomScalar()),
    });
    await compliance.register(pks);

    // find related key (-x, y)
    const PRIME =
        21888242871839275222246405745257275088548364400416034343698204186575808495617n;
    const ORDER =
        21888242871839275222246405745257275088614511777268538073601725287587578984328n;
    const relatedX = PRIME - user.publicKey[0];
    const realPriv = await crypto.formatPrivateKeyForBabyJub(user.privateKey);
    const relatedPriv = ORDER - realPriv;
    const relatedUser = {
        privateKey: relatedPriv,
```

```
        publicKey: [relatedX, user.publicKey[1]] as Point,
    };

    // share secret
    let { input, proofPayload } = await makeShareSecretParams(
        user,
        compliance,
        crypto,
        false // hack
    );

    // proof contains the normal user key (x, y),
    // but public key (which contributes to the AML pubkey) is (-x, y)
    await expect(
        compliance.shareSecret(input, proofPayload, [
            relatedUser.publicKey[0],
            relatedUser.publicKey[1],
        ])
    ).to.be.reverted;
});
```

with the following modifications in `packages/crypto/src/secretSharing.ts` to allow direct use of the private key:

```
async function secretSharing(
   secretSharingArtifacts: Artifacts,
   groth16: any,
   utils: Utils,
+  hack?: boolean,
 ): Promise<SecretSharingProof> {
+  let realPrivKey = privateKey;
+  if (!hack) {
+    realPrivKey = await utils.formatPrivateKeyForBabyJub(privateKey);
+  }
+
   // construct the polynomial for the privateKey
   indexes.forEach((el) => assert(el !== 0n)), 'invalid index';
-  const polynomial = constructPolynomial(threshold,
-    await utils.formatPrivateKeyForBabyJub(privateKey),
-    utils,
-  );
+  const polynomial = constructPolynomial(threshold, realPrivKey, utils);
```

and the following changes in `packages/contracts/test/compliance.test.ts` to enable the tweak:

```
-const makeShareSecretParams = async (sender: KeyPair, compliance: any, crypto: Utils,
index?: bigint) => {
+const makeShareSecretParams = async (sender: KeyPair, compliance: any, crypto: Utils,
hack?: boolean, index?: bigint) => {
```

Recommendation. We recommend avoiding using an ambiguous representation of the public key in general. Either use the compressed format (which packs the sign of x in the MSB of y), or use the uncompressed format.

04 - Lack of clear state transitions in the compliance contract

Compliance.sol

Description. There are several *implied* state transitions in the compliance contract. For example, the contract could have been uninitialized (with an identity AML public key $(0, 1)$), or initialized but with not enough peers (due to the threshold set at 5), or initialized with enough peers but new members are still being added, and so on.

Setup flow. In the setup flow, new public keys can be added by the owner of the smart contract:

1. The function `register()` takes an arbitrary-length array of public keys (as well as an index for each public key, used by the secret sharing scheme).
2. When everyone is registered (by potentially multiple calls to `register()`), then each committee member should call `shareSecret()`. This will split their own secret between everyone else (in a verifiable way) and add their contribution to the AML public key.
3. In the case were the number of registered keys is greater than `SHARE_SIZE` (currently set at 11), then the previous call to `shareSecret()` only encrypted shares to some of the participants, and thus calls to `shareSecretForOne()` are necessary to distribute shares to the rest of the committee.

None of the calls described are being tracked. It is thus possible for the AML public key to be used before everyone has had a chance to share their secret. It is also possible that the committee won't be large enough compared to the threshold (currently set at 5), or that some members won't have received everyone's shares and thus will have trouble contributing in threshold decrypting transactions that happened during that period.

New member flow. New members can be added via the following steps:

1. New keys can be registered by the smart contract owner at any point in time.
2. A new member willing to join the committee will have to call `shareSecret()` to share their own secret with the current committee. Potentially followed by several calls to `shareSecretForOne()`.
3. Then, the rest of the committee will have to call `shareSecretForOne()` to make sure that the new member also has everyone's shares.

Here again, the new AML public key (after step 2) can be used in spite of step 3 not having been executed (or in spite of the new member not having distributed shares to everyone).

Removing a member flow. Members can be removed from the committee via the following steps:

1. The smart contract owner can remove the member by calling `remove()`. This will remove their share in the AML public key.

After executing this step, the committee might be under the `THRESHOLD` (currently set at 5) and thus incapable of decrypting transactions until a new member is added.

Furthermore, the member that was excluded still has access to the shares of others. With enough collusion between dishonest or/and excluded members (assuming that members get kicked out of the committee for acting maliciously), it might be possible for them to recover the AML private key.

Pending keys. In the previous flows, we omitted an important aspect of the compliance smart contract: if there are too many registered keys, public keys get added to an array of pending keys. This notion of pending keys

impacts all aspects of the state transitions. For example, when a registered key is removed, it is replaced with a pending key (in FIFO order).

It is not clear how the smart contract owner should deal with pending keys. For example, If the maximum number of committee members is increased (via increasing `_maxRegisteredKeys`) how can the owner transition pending keys to newly available seats in the committee? It seems like the only way is to call `remove()` on a register key, which might have unwanted additional effects. Another example, what if the owner decreased `_maxRegisteredKeys` instead, and then needed to remove some committee members? Calling `remove()` will actually not `remove()` a member, it will replace them with the first pending user.

Recommendation. Now that we have stated the different issues with the lack of explicit encoding of the state transitions in the smart contract, we discuss some possible solutions:

- Add a global boolean that indicates if the contract has been initialized. make sure that the SMASP contract cannot use an uninitialized contract by checking that boolean. Have the owner manually flip that boolean, or have it be flipped automatically when enough keys have been registered and shared.
- Think about removing the notion of pending keys. It might be an unnecessary feature that adds complexity and more surface area for bugs.
- Think about ways to rotate shares when members are being removed for malicious behavior. Rotating shares could either be done by prompting current members from rejoining under a different keypair, or by including an epoch number that increases when members leave and prompts a new resharing.

05 - Committee members might be able to contribute several times

Compliance.sol

Description. A user attempting to join the compliance committee is expected not to call `shareSecret()` several times. This is checked in the `shareSecret()` function by ensuring that the y-coordinate of the commitment to the first coefficient of the user's polynomial (the user key) is 0:

```
require(yPolynomialPoints[yPublicKey][0] == 0, "Already shared secret");
```

It is not clear at present, if this check is impossible to pass for a malicious user. If it is possible, then the user could join the committee several times. While this would not allow them to contribute to the AML public key several times (due to an additional check with `isPublicKeyAdded`), this will emit several events with potentially different polynomial commitments and encrypted shares. This, in turn, could potentially muddle the threshold decryption of transactions later on.

Passing this check seems possible if either the malicious user can find the discrete logarithm of one of the two points with coordinate $y = 0$, which is highly unlikely, or if the circuit can interpret the point $(0, 0)$ as the point at infinity, which is more likely.

For example, the solidity implementation of the Baby Jubjub curve used currently handles $(0, 0)$ as the point-at-infinity (likely for compatibility with applications that encode the identity as $(0, 0)$). See `CurveBabyJubJub.sol`:

```
function pointAdd(uint256 _x1, uint256 _y1, uint256 _x2, uint256 _y2) internal view returns
(uint256 x3, uint256 y3) {
    if (_x1 == 0 && _y1 == 0) {
        return (_x2, _y2);
    }

    if (_x2 == 0 && _y1 == 0) {
        return (_x1, _y1);
    }
```

We were not able to ensure that this was not the case as the circuits were not shared with the consultants. If it is the case, the malicious user would simply have to use one of the valid points that has $y = 0$ (which is easy to compute), and then submit a proof that uses the secret share 0.

A secret share of 0 would mean that the first coefficient of the user's polynomial is 0, which in turn would mean that the commitment to that coefficient would be the point at infinity. If that point at infinity can be encoded as $(0, 0)$ then it will match the registered key's y coordinate as well as the check mentioned above.

Recommendation. As discussed in **finding #2**, do not use an ambiguous way to refer to a public key. Addressing a public key with its uncompressed coordinates or its compressed coordinates is non-ambiguous, while using only the y-coordinate is ambiguous.

Additionally, use stricter methods to check if a member has already called `shareSecret()`. For example, by using a map from public keys to a boolean in the global state.

06 - Index might not be unique in compliance setup

Compliance.sol

Description. Members of the compliance committee are registered through the `register` function of the compliance contract. (Note that only the contract owner can call the `register` function.) For each public key being registered, an index is also associated with it. This is necessary for the distributed key generation to work, as each member needs to receive evaluations at the same point (or index).

```
function register(PublicKey[] memory pk) public onlyOwner {
    // TRUNCATED...

    indexes[pk[i].yPublicKey] = pk[i].index;
```

The problem is that the owner might use the same index twice, which would result in a smaller committee than expected. Two members might act as one, in some sort.

Recommendation. It might be a good idea to use a counter to compute the next index, and increment it as part of the logic, or check that the index is not already used during registration.

It also seems like an acceptable risk if the owner is careful enough.

07 - Solidity version is outdated

*

Description. In `/packages/contracts/hardhat.config.ts` the version of the Solidity compiler to use in the project is fixed:

```
solidity: {
    version: "0.8.9",
```

The latest version of the Solidity compiler at the time of this writing is 0.8.20 (**https://github.com/ethereum/solidity/releases**), which was released on May 10, 2023. The version currently used by Silent Protocol is from September 29th, 2021 (**https://github.com/ethereum/solidity/releases/tag/v0.8.9**), which is severely outdated.

Recommendation. As there's been a number of bug fixes since version 0.8.9, we recommend Silent Protocol to upgrade the version of the Solidity compiler to the most recent one.

09 - Uniswap contract version is outdated

SMASP.sol

Description. Silent core makes use of the uniswap contract **UniswapV2Router01** in order to calculate user fees in Silent token.

The Uniswap documentation warns projects to migrate to the new version of the contract:

> *"UniswapV2Router01 should not be used any longer, because of the discovery of a low severity bug and the fact that some methods do not work with tokens that take fees on transfer. The current recommendation is to use UniswapV2Router02."*

Recommendation. While the bug affecting the version of the dependency used doesn't seem to affect Silent Core's use of the smart contract, the mention that "some methods do not work with tokens that take fees on transfer" might be enough of a reason to upgrade the dependency.

0b - Remove function lacks explicit preconditions

Compliance.sol

Description. The remove function of the compliance smart contract does not explicitely handle the case where the given public key is not registered (and thus cannot be removed).

While we did not find ways to exploit this (the execution reverts later), it seems like the code forgets to explicitely check that the public key is registered before trying to remove it.

The function with added comments:

```
function remove(uint256[2] memory publicKey) public onlyOwner {
    // this will return 255 if empty
    uint8 index = getRegistrationIndexOfPublicKey(publicKey[1]);

    uint256 _firstPendingKeyIndex = firstPendingKeyIndex;

    if (pendingKeys.length > _firstPendingKeyIndex) {
        // this will error as index is out of bound
        registeredKeys[index] = pendingKeys[_firstPendingKeyIndex];

        // TRUNCATED...
    } else {
        // this will error as index is out of bound
        registeredKeys[index] = registeredKeys[registeredKeys.length - 1];

        // TRUNCATED...
    }

    // TRUNCATED...
}
```

Recommendation. We recommend adding an explicit error at the top of the function in case `getRegistrationIndexOfPublicKey` returns the value 255 (which stands for unregistered public key).