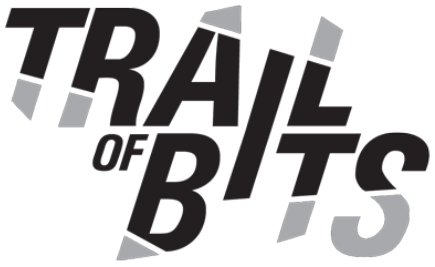

AZTEC

Security Assessment

October 9th, 2019

Prepared For:
Arnaud Schenk | *AZTEC*
arnaud@aztecprotocol.com

Prepared By:
Ben Perez | *Trail of Bits*
benjamin.perez@trailofbits.com

David Pokora | *Trail of Bits*
david.pokora@trailofbits.com

James Miller | *Trail of Bits*
james.miller@trailofbits.com

Will Song | *Trail of Bits*
will.song@trailofbits.com

Paul Kehrer | *Trail of Bits*
paul.kehrer@trailofbits.com

Alan Cao | *Trail of Bits*
alan.cao@trailofbits.com

Executive Summary

From September 9th through September 27th, 2019, AZTEC engaged with Trail of Bits to review the security of the AZTEC protocol. Trail of Bits conducted this assessment over the course of six person-weeks with two engineers using commit hash c3f49df5 for the AztecProtocol/AZTEC repository.

The assessment was dedicated to a review of the AZTEC protocol and codebase. Trail of Bits continued a cryptographic review of the documentation and the AZTEC whitepaper, and documented and assessed any deviations from these specifications (Appendix C). In addition, we reviewed the AZTEC smart contracts to detect any unsafe, low-level Solidity behaviors. Finally, we integrated Echidna into the Solidity smart contracts and provided guidance for future improvements to Solidity testability (Appendix B) to improve coverage and detect unsafe behavior.

Over the course of the audit, Trail of Bits discovered two high-severity issues (TOB-AZT-010 and TOB-AZT-013), both of which were found to have low exploitation difficulty. TOB-AZT-010 can lead to an adversary arbitrarily increasing their balance, and TOB-AZT-013 can lead to an adversary controlling permissions for note spending. Trail of Bits also discovered three medium-severity issues (TOB-AZT-001, TOB-AZT-009, and TOB-AZT-015) concerning potential non-standard token behavior, null administrator addresses, and invalid contracts, respectively. Additionally, two low-severity issues were reported. The cryptographic low-severity issue, TOB-AZT-011, was found to be of high difficulty to exploit, relying on low-probability events. The final low-severity issue, TOB-AZT-016, resulted from not using `SafeMath` to prevent integer overflow. The remaining two issues, TOB-AZT-012 and TOB-AZT-014, were found to be of undetermined and informational severity, respectively. For these, we do not report any immediate threats, but we believe they merit attention.

During the week of September 30th, Trail of Bits also dedicated more resources to the manual review of the AZTEC smart contracts. The additional review of the smart contracts led to the discovery of eight additional data validation findings, as well as an update to a previous finding, TOB_AZT_015. One of these findings, TOB_AZT_021, was reported to be of high severity, as it could lead to the contract's owners losing all the funds in their wallet. We report one finding, TOB_AZT_023, to be of medium severity, as it incorrectly results in owners withdrawing their entire wallet balance. Five of the eight findings were reported to be of low severity. One of these findings, TOB_AZT_018, was already discovered by AZTEC; they have supplied a fix, which we have verified and documented in Appendix D. The last finding, TOB_AZT_022, was reported to be of undetermined severity, as it could lead to unexpected behaviors.

In almost all of the cryptographic findings discovered by Trail of Bits, the implementation deviated from the protocol specified in the documentation. We do not report any cryptographic findings in the documentation itself.

We recommend that AZTEC adhere as closely as possible to their documentation in their implementations going forward. We also encourage AZTEC to stay informed of issues that are found in other Solidity contracts. Lastly, we recommend that AZTEC integrate our tooling, Echidna, to continue to expand test coverage. Specifically, we encourage AZTEC to supplement their Solidity to allow for more coverage with Echidna.

During the week of September 30th, Trail of Bits performed a review of the fixes proposed by AZTEC for the issues in this report. The fixes submitted by AZTEC were either fixed or their risk was accepted. For more details on the review of AZTEC's fixes, see Appendix D.

Project Dashboard

Application Summary

Name	AZTEC
Version	AztecProtocol/AZTEC commit: c3f49df5 AztecProtocol/Setup commit: 230a1d8a
Type	C++, JavaScript, Solidity, TypeScript
Platforms	Ethereum

Engagement Summary

Dates	August 26th through September 27th 2019
Method	Whitebox
Consultants Engaged	2
Level of Effort	10 person-weeks

Vulnerability Summary

Total High-Severity Issues	3	■■■
Total Medium-Severity Issues	4	■■■■
Total Low-Severity Issues	7	■■■■■■■
Total Informational-Severity Issues	1	■
Total Undetermined-Severity Issues	2	■■
Total	17	

Category Breakdown

Cryptography	3	■■■
Data Validation	13	■■■■■■■■■■■■■
Undefined Behavior	1	■
Total	17	

Engagement Goals

The AZTEC protocol uses commitment functions and zero-knowledge proofs to define a confidential transaction protocol. Since the transactions are confidential, the exact balances of the digital assets are secret, but their validity is established with range proofs. The design of the protocol was aimed at reducing the complexity of constructing and verifying proofs. Specifically, AZTEC aims to minimize the cost of Ethereum gas for verification. To achieve this, AZTEC presents an optimized proof system that allows for several proofs to be validated with only a single elliptic curve pairing operation.

AZTEC sought to assess the security of their protocol, the AZTEC protocol. This portion of the engagement was scoped to provide a cryptographic and security assessment of the `ACE` and `ERC1724` smart contracts. Assessment of these contracts coincided with assessment of the corresponding JavaScript packages, including aztec.js and secp256k1. These assessments took the form of manual review and dynamic test coverage using Echidna.

Specifically, we sought to answer the following questions:

- Is the AZTEC protocol specified in the AZTEC whitepaper cryptographically secure?
- Does the implementation of the AZTEC protocol in the `ACE` and `ERC1724` contracts comply with the whitepaper?
- If the implementation does not comply with its corresponding documentation, does this introduce any vulnerabilities?
- Are there any unsafe, low-level Solidity behaviors? Does the use of the `delegatecall` pattern in Solidity introduce any vulnerabilities?
- Are there any flaws in how permissions and modifiers are used in Solidity?
- Can we use dynamic testing to detect unsafe behavior in Solidity?

Coverage

`ACE` contracts. Manual review was first performed on the AZTEC whitepaper. Once that review was complete, Trail of Bits assessed how well the `ACE` contracts adhered to the whitepaper specifications. We also reviewed the Solidity assembly and other areas of the codebase that can lead to dangerous behavior, like the `delegatecall` pattern and the use of `SafeMath`. Review of the `ACE` contracts coincided with a review of the `aztec.js` package, as this package is used to interact with the `ACE` contracts.

`ERC1724` contracts. Trail of Bits also dedicated manual review to the `ERC1724` contracts. These contracts use a signature scheme that is mentioned in the documentation but not in great detail; for example, features like `confidentialApprove` were not mentioned in the supplied documentation. As a result, we performed a cryptographic review of the digital

signatures used throughout these contracts. In addition, the Solidity code was again reviewed for potential dangerous behavior. Review of these contracts coincided with a review of the `secp256k1` signature package.

Recommendations Summary

Short Term

❑ **Add support for popular ERC20 tokens with incorrect/missing return values.** Doing so prevents non-standard ERC20 tokens from failing unexpectedly with AZTEC. TOB-AZT-001

❑ **Always perform zero-address checks when setting up permissions.** If this value is set to zero, the contract cannot be administered. TOB-AZT-009

❑ **Validate output notes in the `Swap` protocol by using the same elliptic curve pairing verification check as the other Join-Split protocols.** Without this verification check, adversaries can raise their balance arbitrarily. TOB-AZT-010

❑ **Add an additional value as input into the hash function at each iteration in the JoinSplit verifier to avoid the security reductions associated with nested hashes.** Nested hashes reduce the image space and lower the security margin of the hash function. TOB-AZT-011

❑ **Measure the gas savings from Solidity compiler optimizations, and carefully weigh them against the possibility of an optimization-related bug.** Solidity compiler optimizations have had multiple security-related issues in the past. TOB-AZT-012

❑ **Update `confidentialApprove` to tie the `_status` value into the signature used to give and revoke permission to spend.** An attacker can modify permissions without authorization because the signature is not tied to `_status`. TOB-AZT-013

❑ **Maintain state to prevent replay of previous signatures with `confidentialApprove`.** An attacker can replay previous signatures to reinstate revoked permissions. TOB-AZT-013

❑ **Add checks in the `NoteRegistryManager` to prevent unsafe `scalingFactor` values.** Accidental input of unsafe values could result in a Denial of Service. TOB-AZT-014

❑ **Prevent calling invalid contracts in `MultiSigWallet` by adding checks in `addTransaction` and `executeTransaction` or `external_call`.** When a transaction calls a self-destructed contract, the transaction will still execute without the contract behaving as expected. TOB-AZT-015

❑ **Use `SafeMath` to avoid overflow in time locking contracts.** Overflow can cause checks to improperly pass. TOB-AZT-016

❑ **Update `noteAccess` in `ZkAssetBase.sol` to be a double mapping that maps an `address` and `noteHash` to a boolean.** This double mapping would prevent malicious parties from altering other parties' access to notes. TOB_AZT_017

❑ **Refactor `ZkAssetMintableBase` to call the `supportsProof` method inside the `confidentialTransferFrom` method.** Calling `supportsProof` ensures that the submitted proof is of the valid type. TOB_AZT_018

❑ **Adjust the `NoteRegistryManager` to use the correct value when accessing `validatedProofs`.** The code currently uses the wrong value and does not properly invalidate proofs, which could lead to proofs being reused. TOB_AZT_019

❑ **Document the fact that fallback functions and functions with similar function IDs can be executed when calling `emergencyExecuteInvalidateProof`.** Users should be aware that some functions can be called here to bypass the `multiSigWallet` time lock. TOB_AZT_020

❑ **Add the `notNull` modifier to `replaceOwner` in `MultiSigWallet.sol`.** Without this modifier, it is possible for a wallet's funds to be trapped permanently. TOB_AZT_021

❑ **Check the return value of all external calls.** Failure to perform these checks could lead to unexpected behaviors in the case of incorrect execution. TOB_AZT_022

❑ **Update the `withdraw` method in `Ace.sol` to withdraw `_amount` and not the account's balance.** Any call to `withdraw` will transfer the entire balance. TOB_AZT_023

❑ **Consider using a lower bound for the fee computation in `Chargeable.sol`.** Without this lower bound, miners can avoid paying the fee for transactions. TOB_AZT_024

Long Term

❑ **Carefully review common misuses of the ERC20 standard.** Slight deviations from this standard have led to subtle bugs with interoperability in the past. TOB-AZT-001

❑ **Review invariants within all components of the system and ensure these properties hold.** Consider testing these properties using a property-testing tool such as Echidna to increase the probability of detecting dangerous edge cases. TOB-AZT-009

❑ **Whenever developing contracts for verifying Join-Split protocols, ensure that there is always an elliptic curve pairing check.** Consider property-based testing with tools such as Echidna in these types of validators and other cryptographic protocols. TOB-AZT-010

❑ **Check that none of the x_i values during verification repeat or are equal to `0`.** Without these checks, invalid data may pass verification. TOB-AZT-011

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity.** As the Solidity compiler matures, these optimizations should stabilize. TOB-AZT-012

❑ **Adjust the signature verified in `confidentialApprove` so that it is not replayable.** Mixing a nonce and the status type with the signature allows for minimal state while preventing both replay and mutability. TOB-AZT-013

❑ **Ensure all call sites where `SafeMath` is used are carefully verified.** Improper use of `SafeMath` can trap certain state transitions. TOB-AZT-014

❑ **Monitor issues discovered in code that is imported from external sources.** Contracts like `MultiSigWallet` are commonly used and can have known vulnerabilities that developers are not aware of. TOB-AZT-015

❑ **Document the purpose of each component of the codebase and ensure that all code is being used.** The purpose of `noteAccess` is unclear, but if it has a significant intended purpose in the codebase, the issue reported for it could be of high severity. TOB_AZT_017

❑ **Add Slither to your CI, or use crytic.io.** These tools can automatically detect potentially bad Solidity behaviors (e.g., not verifying return values) and review correct inheritance behaviors. TOB_AZT_018, TOB_AZT_022

❑ **Consider adding a whitelist of destinations for which `emergencyExecuteInvalidateProof` can be used.** Doing so could prevent unexpected functions from being executed. TOB_AZT_20

❑ **Be aware that miners can control most of the block parameters and avoid relying on the block information.** Miners can manipulate these parameters to perform unintended behaviors, like avoiding transaction fees. TOB_AZT_024

❑ **Consider expanding your Solidity code to allow for better property-based fuzzing.** The current mix of JavaScript and Solidity makes use of fuzzing and property testing tools more difficult; however, using these tools can detect unintended and unsafe behavior. TOB_AZT_019, TOB_AZT_021, TOB_AZT_023, Appendix B

❑ **Ensure the documentation accurately reflects the protocol as implemented.** Deviations between specification and implementation can lead to correctness bugs when other developers attempt to create their own interoperable systems. Appendix C

Findings Summary

#	Title	Type	Severity
1	IERC20 is incompatible with non-standard ERC20 tokens	Data Validation	Medium
9	Missing check for address(0) in constructor of AdminUpgradeabilityProxy	Data Validation	Medium
10	Missing elliptic curve pairing check in the Swap Validator	Cryptography	High
11	Using nested hashes reduces security	Cryptography	Low
12	Solidity compiler optimizations can be dangerous	Undefined Behavior	Undetermined
13	Replay attack and revocation inversion on confidentialApprove	Cryptography	High
14	scalingFactor allows values leading to potentially undesirable behavior	Data Validation	Informational
15	Lack of contract existence check on low-level calls will lead to unexpected behavior	Data Validation	Medium
16	Time locking can overflow	Data Validation	Low
17	Anyone can overwrite note access	Data Validation	Low
18	ZkAssetMintableBase and inherited contracts do not check for proofs supported	Data Validation	Low
19	Incorrect mapping key leads to incorrect records cleaned	Data Validation	Low

20	ACEOwner allows execution of unexpected functions through the emergency function	Data Validation	Low
21	The 0x0 address can become an owner through replaceOwner	Data Validation	High
22	Lack of return value check might lead to unexpected behaviors	Data Validation	Undetermined
23	ACE.withdraw will empty the contract	Data Validation	Medium
24	Miners can avoid paying the fee	Data Validation	Low

1. IERC20 is incompatible with non-standard ERC20 tokens

Severity: Low
Type: Data Validation
Target: `IERC20.sol`

Difficulty: Medium
Finding ID: TOB-AZT-001

Description
`IERC20` is meant to work with any ERC20 token. Several high-profile ERC20 tokens do not correctly implement the ERC20 standard. Therefore, `IERC20` will not work with these tokens.

The ERC20 standard defines three functions, among others:

- `approve(address _spender, uint256 _value) public returns (bool success)`
- `transfer(address _to, uint256 _value) public returns (bool success)`
- `transferFrom(address _from, address _to, uint256 _value) public returns (bool success)`

Several high-profile ERC20 tokens do not return a boolean on these three functions. Starting from Solidity 0.4.22, the return data size of external calls is checked. As a result, any call to `approve`, `transfer`, or `transferFrom` will fail for ERC20 tokens that implement the standard incorrectly.

Examples of popular ERC20 tokens that are incompatible include BinanceCoin, OmiseGo, and Oyster Pearl.

Exploit Scenario
Bob creates a note registry, providing a `linkedTokenAddress` pointing to a non-standard ERC20 token that does not implement the correct interface for `transfer`/`transferFrom`. Upon updating the state of the note registry, a call to the transfer methods will cause a revert.

Recommendation
Short term, consider adding support for popular ERC20 tokens with incorrect/missing return values. This could be achieved with a whitelist for such tokens that should use an alternative interface. Alternatively, a carefully crafted low-level call would prevent a revert and allow for manual validation of return data.

Long term, carefully review the usage and issues of the ERC20 standard. This standard has a history of misuses and issues.

References

- Missing return value bug—At least 130 tokens affected
- Explaining unexpected reverts starting with Solidity 0.4.22

9. Missing check for address(0) in constructor of AdminUpgradeabilityProxy

Severity: Medium
Type: Data Validation
Target: `AdminUpgradeabilityProxy.sol`

Difficulty: High
Finding ID: TOB-AZT-009

Description
In `BaseAdminUpgradeabilityProxy`, the `changeAdmin` function performs a zero-address check before calling `_setAdmin()`.

```
function changeAdmin(address newAdmin) external ifAdmin {
        require(newAdmin != address(0), "Cannot change the admin of a proxy to the zero address");
        emit AdminChanged(_admin(), newAdmin);
        _setAdmin(newAdmin);
    }
```

Figure 9.1: `changeAdmin` function in `BaseAdminUpgradeabilityProxy`.

However, the constructor of `AdminUpgradeabilityProxy` calls `_setAdmin()` without performing a zero-address check.

```
constructor(address _logic, address _admin, bytes memory _data) UpgradeabilityProxy(_logic, _data) public payable {
        assert(ADMIN_SLOT == bytes32(uint256(keccak256('eip1967.proxy.admin')) - 1));
        _setAdmin(_admin);
    }
```

Figure 9.2: The constructor for `AdminUpgradeabilityProxy`.

If the constructor for `AdminUpgradeabilityProxy` is called with `_admin` set to zero, then the contract will be un-administrable.

Exploit Scenario
The AZTEC deployment system has an implementation error and mistakenly sets the admin address of `AdminUpgradeabilityProxy` to zero.

A malicious internal user at AZTEC sabotages the setup procedure and uses their administrative privileges to set the admin address in `AdminUpgradeabilityProxy` to zero.

Recommendation
Short term, always perform zero-address checks when setting up permissions.

Long term, review invariants within all components of the system and ensure these properties hold. Consider testing these properties using a property-testing tool such as Echidna.

10. Missing elliptic curve pairing check in the Swap Validator

Severity: High
Type: Cryptography
Target: `Swap.sol`

Difficulty: Low
Finding ID: TOB-AZT-010

Description
The Swap protocol does not perform an elliptic curve pairing check to validate its output notes. The protocol performs all of the other checks of the Join-Split protocol, but without the elliptic curve pairing, the validator can be tricked into validating output notes with invalid commitments.

Exploit Scenario
Any adversary can easily exploit this by submitting an invalid commitment that is not computationally binding. For example, an adversary can submit the following invalid commitment: $(\gamma, \sigma) = (1, h^a)$. This commitment can be used to commit to `(k,a)` for any `k` value.

An adversary can submit this commitment as an output note to the Swap protocol, where they choose the `k` value that will make the validator verify this proof (this `k` value will just be the `k` value of the input note). Since there is no elliptic curve pairing check, the validator will accept this as valid.

Once accepted by the validator, this malicious note will now be on the Note Registry, where it can be input to any Join-Split protocol. All of the Join-Split protocols will assume that the input notes have already been verified, so they will not detect it as malicious. Since this commitment can commit to any `k` value, an adversary can make this `k` value much higher than the value in the Swap protocol, successfully raising their balance for free.

Recommendation
Short term, validate output notes by using the same elliptic curve pairing verification check as the other Join-Split protocols.

Long term, whenever developing contracts for verifying Join-Split protocols, ensure that there is always an elliptic curve pairing check. Consider property-based testing with tools such as Echidna in these types of validators and other cryptographic protocols.

11. Using nested hashes reduces security

Severity: Low
Type: Cryptography
Target: `ACE/validators`

Difficulty: High
Finding ID: TOB-AZT-011

Description
The validator contracts implement an optimized Join-Split verification that uses random x_i values to verify a proof with only one elliptic curve pairing operation. To compute the `i`-th `x` value, the validator computes `i` hashes on the same input. Using nested hashes in this manner can be shown to be less secure than using the regular hash function. For instance, for a hash function `H`, it can be shown that it is `n` times easier to invert $H^n(x)$ than `H(x)` for any `x` (here, $H^n(x)$ refers to computing the hash of `x`, `n` times). For more information, see "A Graduate Course in Applied Cryptography" by Boneh and Shoup, section 18.4.3.

Exploit Scenario
With a weakened hash function generating random values, it is easier for an adversary to discover inputs that generate problematic random values. For instance, if an adversary could produce inputs that hash to the value of `0`, they can cause the validator to validate invalid output notes. Additionally, if an adversary could cause the hash function to produce an identical (x_i, x_j) pair (with `i` not equal to `j`), they can also cause the validator to validate invalid output notes.

Recommendation
Short term, add an additional value as input into the hash function at each iteration, rather than solely inputting the output of the previous iteration.

Long term, check that none of the x_i values repeat or are equal to `0` to prevent similar attacks from happening.

12. Solidity compiler optimizations can be dangerous

Severity: Undetermined
Type: Undefined Behavior
Target: `Solidity`

Difficulty: Low
Finding ID: TOB-AZT-012

Description
There have been several bugs with security implications related to optimizations. Moreover, optimizations have changed in the recent past. Solidity compiler optimizations are disabled by default. It is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe. Moreover, the Common Subexpression Elimination (CSE) optimization procedure is "implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function." Similar code in other large projects have resulted in bugs.

There are likely latent bugs related to optimization, and/or new bugs that will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations causes an unexpected security vulnerability in a contract.

Recommendation
Short term, measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

13. Replay attack and revocation inversion on confidentialApprove

Severity: High
Type: Cryptography
Target: `ZKAssetBase.sol, aztec.js/src/signer/index.js`

Difficulty: Low
Finding ID: TOB-AZT-013

Description
The `confidentialApprove` method is used to allow third parties to spend notes on the note owner's behalf. In the `confidentialApprove` method, there is a call to `validateSignature`, which verifies that the signature giving (or revoking) permission to spend is valid and was signed by the owner. However, when verifying the signature, the `_status` (indicating giving or revoking of permission) is not actually tied to the signature (see figure 13.1), so the same signature used to revoke permission can be used to restore permission. This problem is compounded because there is no mechanism in place to detect the resubmission of previously used signatures.

```
function confidentialApprove(
        bytes32 _noteHash,
        address _spender,
        bool _status,
        bytes memory _signature
    ) public {
        ( uint8 status, , , ) = ace.getNote(address(this), _noteHash);
        require(status == 1, "only unspent notes can be approved");
        bytes32 _hashStruct = keccak256(abi.encode(
                NOTE_SIGNATURE_TYPEHASH,
                _noteHash,
                _spender,
                status
        ));

        validateSignature(_hashStruct, _noteHash, _signature);
        confidentialApproved[_noteHash][_spender] = _status;
    }
```

Figure 13.1: The `confidentialApprove` method in `ZKAssetBase.sol`. The `_status` value is not included in the `_hashStruct` used to verify the signature.

Exploit Scenario
This can be exploited to wrongfully give or revoke permissions. Say an owner decides to revoke permission that was previously given to a third party. That party can resubmit either the original signature giving permission or the latest signature revoking permission (where they switch the permission back to true) and wrongfully reinstate their permission to spend notes.

Further, another malicious party (who is neither the owner nor third party) can revoke any permission given to third parties. If an owner submits a signature giving permission to a third party, this malicious party can easily resubmit this signature and switch the `_status` to `false`. Again, since `_status` is not tied to the signature, they will accept this malicious submission.

Recommendation
Short term, update `confidentialApprove` to tie the `_status` value into the signature used to give/revoke permission to spend. In order for valid signatures to work, this will also require updating `signer/index.js` in `aztec.js` to tie this value into the signature (presently, the `_status` value is always set to `true`; see Figure 13.2). In addition, we recommend maintaining state to prevent the replay of previous signatures.

```
signer.signNoteForConfidentialApprove =
  (verifyingContract, noteHash, spender, privateKey) => {
    const domain = signer.generateZKAssetDomainParams(verifyingContract);
    const schema = constants.eip712.NOTE_SIGNATURE;
    const status = true;
    const message = {
        noteHash,
        spender,
        status,
    };

    const { unformattedSignature } = signer.signTypedData(domain, schema, message, privateKey);
    const signature = `0x${unformattedSignature.slice(0, 130)}`;
    return signature;
};
```

Figure 13.2: The `signNoteForConfidentialApprove` method in `signer/index.js`. The `status` value is always set to `true`.

Long term, adjust the signature scheme so it is not detachable in this way, in order to prevent similar replay attacks in the future.

14. scalingFactor allows values leading to potentially undesirable behavior

Severity: Informational
Type: Data Validation
Target: `NoteRegistryManager.sol`

Difficulty: Hard
Finding ID: TOB-AZT-014

Description
The `NoteRegistryManager` contract defines how notes will be managed. When creating a note registry, a `scalingFactor` is defined, which represents the value of an AZTEC asset relative to another asset. Both the `scalingFactor` and `totalSupply`, which represents the total amount of notes, are `uint256` types which are operated on via `SafeMath` functions. When a transaction is performed that requires the supply to be increased (e.g., when another asset is converted into an AZTEC asset), the `scalingFactor` is used to determine how much to increase the supply.

Since the `totalSupply` is a `SafeMath uint256`, the supply will be capped at the maximum `uint256` value. By design, `SafeMath` ensures that once the `totalSupply` reaches this maximum value, every transaction that results in an increase in the `totalSupply` will result in a `revert`. Since the maximum `unit256` value is very large, this will only be problematic when the `scalingFactor` value is set to very small values or `0`. However, there is currently no mechanism in place to prevent potentially unsafe `scalingFactor` values.

Exploit Scenario
A note registry for an asset is incorrectly set up with a very small `scalingFactor` value. A user then employs a few tokens of another asset to cap out the supply of this asset. As a result, no other users can obtain this asset, and all transactions that would increase supply will `revert`.

Recommendation
Short term, add checks in the `NoteRegistryManager` to prevent unsafe `scalingFactor` values from being used.

Long term, ensure all call sites where `SafeMath` is used are carefully verified, as it can trap certain state transitions.

15. Lack of contract existence check on low-level calls will lead to unexpected behavior

Severity: Medium
Type: Data Validation
Target: `MultiSigWallet.sol, Proxy.sol, ACE.sol`

Difficulty: Low
Finding ID: TOB-AZT-015

Description
Transactions are added to the `MultiSig` wallet via `submitTransaction`. Except for a zero check, there is currently no mechanism to verify that the `destination` address corresponds to a valid contract.

```
modifier notNull(address _address) {
    require(_address != address(0x0));
    _;
}

function addTransaction(address destination, uint value, bytes memory data)
    internal
    notNull(destination)
    returns (uint transactionId) {
    // ...
}
```

Figure 15.1: The `addTransaction` *method in* `MultiSigWallet.sol`*.*

Furthermore, contract validity is not checked in `external_call`, and it is documented that the low-level instruction `call` will return `true` even if the target contract does not exist. When `external_call` is executed by `executeTransaction` with an invalid contract, the wallet will mark the transaction as executed.

```
function external_call(address destination,
                       uint value, uint dataLength,
                       bytes memory data) internal returns (bool) {
    bool result;
    assembly {
        let x := mload(0x40)
        let d := add(data, 32)
        result := call(
            sub(gas, 34710),
            destination,
            value,
            d,
            dataLength,
            x,
            0
        )
    }
    return result;
}
```

Figure 15.2: The `external_call` *method that does not contain a validity check.*

This issue also occurs in two other locations:

- `delegatecall` in `Proxy.sol#L39`
- `staticcall` in `ACE.sol#L179`

Note that low-level calls to pre-compiled contracts do not need the code's existence check (such as in `LibEIP712.sol#L128`).

Exploit Scenario
A user of the multisignature wallet may unknowingly submit transactions to a self-destructed contract. The transaction will eventually get executed in `executeTransaction`, and the expected behavior of the wallet will not occur, which can cause unintentional bugs.

Recommendation
Short term, add two checks: one in `addTransaction` to prevent the creation of bad transactions, and one in `executeTransaction` or `external_call` to prevent the race condition of the target contract's self-destruction. Checks for contract existence should be performed prior to all low-level calls to prevent these unexpected behaviors.

Long term, monitor issues discovered with `MultiSigWallet` and any other code imported from external sources.

16. Time locking can overflow

Severity: Low
Type: Data Validation
Target: `MultiSigWalletWithTimeLock.sol`

Difficulty: High
Finding ID: TOB-AZT-016

Description
The `MultiSigWithTimeLock` contract supports a time locking feature with a wallet settable time lock. However, the time lock check is an unbounded addition which can overflow and cause the check to always pass.

```
    modifier pastTimeLock(uint256 transactionId) {
        require(
            block.timestamp >= confirmationTimes[transactionId] + secondsTimeLocked,
            "TIME_LOCK_INCOMPLETE"
        );
        _;
    }
```

Figure 16.1: The `pastTimeLock` method in the `MultiSigWithTimeLock` contract.

Exploit Scenario
Alice and Eve are owners of the `MultiSigWalletWithTimelock`. Alice wishes to set the time lock to be unlocked at a date which is virtually never going to be encountered, allowing the wallet to be abandoned. In this case, Alice submits a transaction to execute changeTimeLock with a large number. Eve knows this will cause an overflow, allowing immediate execution of previously locked transactions. Eve is now able to execute previously locked transactions.

Recommendation
Use `SafeMath` to avoid overflow on sensitive mathematical operations.

17. Anyone can overwrite note access

Severity: Low
Type: Data Validation
Target: `ZkAssetBase.sol`

Difficulty: Low
Finding ID: TOB-AZT-017

Description
`ZkAssetBase.noteAccess` associates an address to a note. The lack of validation allows anyone to replace an existing association.

`noteAccess` is updated in `updateNoteMetaData`:

```
function updateNoteMetaData(bytes32 noteHash, bytes calldata metaData) external {
        // Get the note from this assets registry
        ( uint8 status, , , address noteOwner ) = ace.getNote(address(this), noteHash);
        require(status == 1, "only unspent notes can be approved");


        require(
            noteAccess[msg.sender] == noteHash || noteOwner == msg.sender,
            'caller does not have permission to update metaData'
        );


        address addressToApprove = MetaDataUtils.extractAddresses(metaData);
        noteAccess[addressToApprove] = noteHash;


        emit ApprovedAddress(addressToApprove, noteHash);
        emit UpdateNoteMetaData(noteOwner, noteHash, metaData);
```

Figure 17.1: `updateNoteMetaData` *(*`ZkAssetBase.sol#L292`*)*

There is no validation that the address to be associated accepts the association. In addition, an address can be associated only with one note. As a result, an attacker can remove another address' existing association by associating the address with a controlled note.

We classified the issue as low severity because in the audited codebase, `noteAccess` is never used. If `noteAccess` is meant to have a more important role, this issue would have a higher severity.

Exploit Scenario
Bob calls `updateNoteMetaData` to associate Alice's address with Bob's note. Eve repeatedly calls `updateNoteMetaData` to associate Alice's address with her note. As a result, Eve prevents Alice from being associated with Bob's note.

Recommendation

Short term, use a double mapping for noteAccess of type:

```
mapping(address => mapping(bytes32 => bool))
```

`noteAccess[address][noteHash]` will return a boolean indicating if the address is associated with the note.

Long term, document the goal of each component and ensure that all code is being used.

18. ZkAssetMintableBase and inherited contracts do not check for proofs supported

Severity: Low
Type: Data Validation
Target: `ZkAssetOwnableBase.sol`, `ZkAssetMintableBase.sol`

Difficulty: Low
Finding ID: TOB-AZT-018

Description
An incorrect inheritance schema leads `ZkAssetMintableBase` to miss the check introduced by `ZkAssetOwnableBase`. `ZkAssetOwnableBase.supportsProof` checks that the given proofs are supported by the asset:

```
  // @dev Return whether the proof is supported or not by this asset. Note that we
have
   //      to subtract 1 from the proof id because the original representation is
uint8,
   //      but here that id is considered to be an exponent
   function supportsProof(uint24 _proof) public view returns (bool) {
       (uint8 epoch, uint8 category, uint8 id) = _proof.getProofComponents();
       require(category == uint8(ProofCategory.BALANCED), "this asset only supports
balanced proofs");
       uint8 bit = uint8(proofs[epoch] >> (id.sub(1)) & 1);
       return bit == 1;
   }
```

Figure 18.1: `supportsProof` *(*`ZkAssetOwnableBase.sol#L52-L56`*).*

The contract's owner defines which assets are supported:

```
function setProofs(
      uint8 _epoch,
      uint256 _proofs
  ) external onlyOwner {
      proofs[_epoch] = _proofs;
  }
```

Figure 18.2: `setProofs` *(*`ZkAssetOwnableBase.sol#L36-L41`*).*

`ZkAssetOwnableBase.confidentialTransferFrom` checks for the support before transferring the token:

```
    function confidentialTransferFrom(uint24 _proof, bytes memory _proofOutput) public {
        bool result = supportsProof(_proof);
        require(result == true, "expected proof to be supported");
```

Figure 18.3: `ZkAssetOwnableBase.confidentialTransferFrom` (`ZkAssetOwnableBase.sol#L43-L45`).

`ZkAssetMintableBase` inherits from `ZkAssetOwnableBase` and overrides `confidentialTransferFrom`:

```
    function confidentialTransferFrom(uint24 _proof, bytes memory _proofOutput) public
{
        (bytes memory inputNotes,
        bytes memory outputNotes,
        address publicOwner,
        int256 publicValue) = _proofOutput.extractProofOutput();

        [ … ]

        if (publicValue > 0) {
            emit RedeemTokens(publicOwner, uint256(publicValue));
        }
    }
```

Figure 18.4: `ZkAssetMintableBase.confidentialTransferFrom` (`ZkAssetMintableBase.sol#L113-L160`).

`ZkAssetMintableBase.confidentialTransferFrom` does not call `supportsProof`. As a result, the check is never made, and unsupported proofs can be transferred.

Additionally, `supportsProof` is never called by any version of `confidentialTransfer`. It is unclear whether this is the expected behavior or not.

AZTEC also discovered this issue and implemented a fix:
https://github.com/AztecProtocol/AZTEC/pull/301

Exploit Scenario
Bob is the asset owner. Bob approves Eve to transfer their note. Bob did not validate the note's epoch and expects Eve to be unable to transfer the note. Since the support check is not performed, Eve transfers the note without Bob's full consent.

Recommendation
Short term, refactor `ZkAssetMintableBase.confidentialTransferFrom` to use `super`. Review whether `confidentialTransfer` should call `supportsProof`.

Long term, use Slither and its inheritance printer to review the correct inheritance.

19. Incorrect mapping key leads to incorrect records cleaned

Severity: Low
Type: Data Validation
Target: `NoteRegistryManager.sol`

Difficulty: Low
Finding ID: TOB-AZT-019

Description
Incorrect key access prevents a note's record from being cleaned during an update. `NoteRegistryManager.validatedProofs` keeps track of the proof's state:

```
mapping(bytes32 => bool) public validatedProofs;
```

Figure 19.1: `NoteRegistryManager.sol#L76`.

Once a proof has been validated, the mapping is set to true:

```
bytes32 validatedProofHash = keccak256(abi.encode(proofHash, _proof, msg.sender));
validatedProofs[validatedProofHash] = true;
```

Figure 19.2: `ACE.sol#L183-L184`.

When the note is used, `NoteRegistryManager.updateNoteRegistry` is supposed to invalidate the mapping element:

```
// clear record of valid proof - stops re-entrancy attacks and saves some
validatedProofs[proofHash] = false;
```

Figure 19.3: `NoteRegistryManager.sol#L370-L371`.

However, the key associated with the element when set to true is computed as:

```
keccak256(abi.encode(proofHash, _proof, msg.sender))
```

And when set to false, it is computed as:

```
proofHash
```

As a result, the initial element will never be set to false and the record will not be cleaned.

According to the Behavior implementation, this might lead an attacker to re-use notes. The current Behavior implementation (`Behaviour201907.sol`) does not allow a spent note to be reused. As a result, it is not an immediate risk.

Exploit Scenario

Bob deploys the contracts with a new version of the behavior implementation that does not prevent a double spending attack. As a result, Eve is able to re-use a proof to spend the same note multiple times.

Recommendation
Short term, use `keccak256(abi.encode(proofHash, _proof, msg.sender))` when accessing `validatedProofs` in `NoteRegistryManager.updateNoteRegistry`.

Long term, use a mock Behavior with Echidna or Manticore to ensure that double spending attacks are prevented directly from the `NoteRegistryManager`.

20. ACEOwner allows execution of unexpected functions through the emergency function

Severity: Low
Type: Data Validation
Target: `ACEOwner.sol`

Difficulty: High
Finding ID: TOB-AZT-020

Description

`ACEOwner` emergency functions are meant to bypass the time lock in case of a call to `invalidateProof`. Due to function ID collisions, or fallback functions, more targets are possible.

`emergencyExecuteInvalidateProof` allows a user to execute a transaction without requiring the expected timelock (`pastTimeLock`):

```
    function emergencyExecuteInvalidateProof(uint256 transactionId)
        public
        notExecuted(transactionId)
        fullyConfirmed(transactionId)
    {
        Transaction storage txn = transactions[transactionId];
        bytes memory txData = txn.data;


        // Revert unless method signature in data is signature for
invalidateProof(uint24 _proof)
        assembly {
            switch eq(shr(224, mload(add(txData, 0x20))), 0xcaaaa5d8)
            case 0 {
                revert(0x00, 0x00)
            }
```

Figure 20.1: `ACEOwner.sol#L23-L36`.

The documentation states that only `invalidateProof(uint24 _proof)` can be called through this function. To enforce it, the function ID of the transaction is compared to 0xcaaaa5d8.

As only four bytes are used for Solidity's function ID, collision is possible. As a result, if the destination has a function with a similar function ID, it is possible to call it without necessitating the timelock. Additionally, the destination's fallback function can be also executed.

Exploit Scenario

Eve creates a transaction that is meant to execute the destination's fallback function. Eve uses `0xcaaaa5d8` as their function ID. Eve's transaction is accepted, and it is executed without the required timelock.

Recommendation
Short term, document this behavior to ensure that all the `MultiSigWallet` users are aware that fallback functions can be executed and that function ID collisions can happen.

Long term, consider adding a whitelist of destinations for which `emergencyExecuteInvalidateProof` can be used.

21. The 0x0 address can become an owner through replaceOwner

Severity: High
Type: Data Validation
Target: MultiSigWallet.sol

Difficulty: High
Finding ID: TOB-AZT-021

Description
The constructor and addOwner of the MultiSigWallet prevents the address 0x0 from being an owner. This property is not enforced by replaceOwner. As a result, 0x0 can be the owner of the contract.

The constructor and addOwner check that the owner is not 0x0:

```
require(!isOwner[_owners[i]] && _owners[i] != 0);
```

Figure 21.1: *Constructor, Check for* notNull *owner (*MultiSigWallet.sol#L115*).*

```
function addOwner(address owner)
    public
    onlyWallet
    ownerDoesNotExist(owner)
    notNull(owner)
```

Figure 21.2: addOwner*, Check for* notNull *owner (*MultiSigWallet.sol#L124-L128*).*

replaceOwner does not perform this check. As a result, an incorrect call to replaceOwner can lead 0x0 to be present in the owner list.

Having 0x0 as an owner increases the risk that the number of required confirmations will be greater than the number of valid owners.

Exploit Scenario
Alice and Bob are the two owners of a MultiSigWallet. The MultiSigWallet requires two confirmations per transaction. Bob wants to change their address and send a transaction to replaceOwner. The new address is incorrectly set to 0x0. Alice validates the transaction. The wallet now has 0x0 as the owner, and it is not possible to confirm any further transactions. The wallet's funds are trapped.

Recommendation
Short term, add the notNull modifier to replaceOwner.

Long term, use Echidna and Manticore to ensure that `0x0` can never be the owner of a contract.

22. Lack of return value check might lead to unexpected behaviors

Severity: Undetermined
Type: Data Validation
Target: `All the smart contracts`

Difficulty: Undetermined
Finding ID: TOB-AZT-022

Description
Slither reports the lack of return value check for several external calls. These missing checks might lead to unexpected behaviors in case of incorrect execution.

For example, several calls to `transfer` and `transferFrom` are not checked, which might lead to incorrectly assuming the transfer for tokens that do not revert:

```
ZkAssetMintableBase.confidentialTransfer(bytes,bytes) (ERC1724/base/ZkAssetMintableBase.sol#66-97)
ignores return value by external calls
"ERC20Mintable(address(linkedToken)).mint(address(this),supplementValue.mul(scalingFactor))"
(ERC1724/base/ZkAssetMintableBase.sol#89)
ZkAssetMintableBase.confidentialTransfer(bytes,bytes) (ERC1724/base/ZkAssetMintableBase.sol#66-97)
ignores return value by external calls
"ERC20Mintable(address(linkedToken)).approve(address(ace),supplementValue.mul(scalingFactor))"
(ERC1724/base/ZkAssetMintableBase.sol#90)
ZkAssetMintableBase.confidentialTransferFrom(uint24,bytes)
(ERC1724/base/ZkAssetMintableBase.sol#113-160) ignores return value by external calls
"ERC20Mintable(address(linkedToken)).mint(address(this),supplementValue.mul(scalingFactor))"
(ERC1724/base/ZkAssetMintableBase.sol#141)
ZkAssetMintableBase.confidentialTransferFrom(uint24,bytes)
(ERC1724/base/ZkAssetMintableBase.sol#113-160) ignores return value by external calls
"ERC20Mintable(address(linkedToken)).approve(address(ace),supplementValue.mul(scalingFactor))"
(ERC1724/base/ZkAssetMintableBase.sol#142)
NoteRegistryManager.supplementTokens(uint256) (ACE/noteRegistry/NoteRegistryManager.sol#149-163)
ignores return value by external calls
"registry.linkedToken.transferFrom(msg.sender,address(this),_value.mul(scalingFactor))"
(ACE/noteRegistry/NoteRegistryManager.sol#162)
NoteRegistryManager.transferPublicTokens(address,uint256,int256,bytes32)
(ACE/noteRegistry/NoteRegistryManager.sol#316-347) ignores return value by external calls
"registry.linkedToken.transferFrom(_publicOwner,address(this),_transferValue)"
(ACE/noteRegistry/NoteRegistryManager.sol#336-339)
NoteRegistryManager.transferPublicTokens(address,uint256,int256,bytes32)
(ACE/noteRegistry/NoteRegistryManager.sol#316-347) ignores return value by external calls
"registry.linkedToken.transfer(_publicOwner,_transferValue)"
(ACE/noteRegistry/NoteRegistryManager.sol#342-345)
ZkAssetOwnableTest.callValidateProof(uint24,bytes) (test/ERC1724/ZkAssetOwnableTest.sol#19-21)
ignores return value by external calls
"zkAssetOwnable.ace().validateProof(_proof,msg.sender,_proofData)"
(test/ERC1724/ZkAssetOwnableTest.sol#20)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

Figure 22.1: Slither result (`--detect unused-return`).

Exploit scenario
An ERC20 token not reverting in case of error is used in AZTEC. Eve uses an AZTEC contract in which `transferFrom` is called and returns false. The AZTEC contract considers the call

successful although no token was transferred. As a result, Eve is able to invalidly withdraw these tokens from the AZTEC contract.

Recommendation
Short term, check the return value of all external calls.

Long term, add Slither to your CI, or use crytic.io.

23. ACE.withdraw will empty the contract

Severity: Medium
Type: Data Validation
Target: `ACE.sol (feat-fee-model PR)`

Difficulty: High
Finding ID: TOB-AZT-023

Description
An incorrect transfer value leads `ACE` to be emptied after a call to `withdraw`.

`withdraw(_destination, _amount)` allows the owner to withdraw a given amount to the destination:

```
function withdraw(address payable _destination, uint256 _amount)
    public
    onlyOwner
    returns (bool)
{
    require(_destination != address(0x0), "can not withdraw to 0x0 address");
    require(_amount <= address(this).balance, "can not withdraw more than balance");
    _destination.transfer(address(this).balance);
    return true;
```

Figure 23.1: `withdraw` *(*`ACE.sol#L414-L422`*).*

The transferred value is the contract's balance instead of the given amount. As a result, calling `withdraw` will transfer all the ethers.

This issue only affects the upcoming feat-fee-model PR.

Exploit Scenario
Bob wants to grant Eve 10 ether, and the ACE contract holds 1,000 ethers. Bob calls `withdraw(Eve, 10)`. Eve improperly receives the 1,000 ethers and refuses to give back the 9,990 ethers.

Recommendation
Short term, use `_amount` in the withdrawal transfer.

Long term, use Echidna and Manticore to ensure that `withdraw` works as intended.

24. Miners can avoid paying the fee

Severity: Low
Type: Data Validation
Target: `Chargeable.sol (feat-fee-model PR)`

Difficulty: High
Finding ID: TOB-AZT-024

Description
`Chargeable` adds a fee to several functions. The fee is based on the transaction's gas price. Miners can accept the transaction with a zero gas price. As a result, miners can avoid paying the fee.

The fee is computed in the `fee` modifier:

```
modifier fee(uint24 _proofId) {
       uint256 gasFee = getFeeForProof(_proofId);
       uint256 feeExpected = gasFee.mul(uint256(tx.gasprice));
       emit TxFee(feeExpected);
       require(msg.value >= feeExpected, "msg.value has insufficient associated fee");
```

Figure 24.1: fee modifier (`Chargeable.sol#L17-L21`*).*

If `tx.gasprice` is zero, the expected fee will be zero.

Exploit Scenario
Eve is a miner. Every time she mines a new block, she adds an AZTEC transaction with a gas price of zero. As a result, Eve avoids paying the fee.

Recommendation
Short term, consider using a lower bound for the fee computation.

Long term, be aware that miners can control most of the block parameters and avoid relying on the block information.

Reference
- **Zero Gas Price Transactions — what they do, who creates them, and why they might impact Scalability**

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Solidity Testability

Trail of Bits has developed Echidna, an EVM fuzzer and we strongly recommend its use on AZTEC's Solidity code. Echidna will automatically generate test cases for the exposed ABI of the Solidity contract that attempt to violate defined security properties.

For example, Echidna would find the following contract falsifiable with two sufficiently large `uint256` values. Methods that begin with `echidna_*` easily define security properties.

```
contract SafeAdd {
  bool failed = false;
  function try_add(uint256 a, uint256 b) public {
    uint256 c = a + b;
    if (c < a || c < b) {
      failed = true;
    }
  }
  function echidna_add() public returns (bool) {
    return !failed;
  }
}
```

Figure D.1: An example contract.

To demonstrate its use, we wrote an Echidna property for TOB-AZT-009. Echidna is able to fuzz the exposed ABI and verify that none of the functions we can call with the contract are able to set the admin address to 0. It is a very short contract, detailed below.

```
pragma solidity ^0.5.0;
import './AdminUpgradeabilityProxy.sol';
contract AdminUpgradeTest is AdminUpgradeabilityProxy {
  constructor() AdminUpgradeabilityProxy(address(this), address(this), "") public { }
  function echidna_admin() public returns (bool) {
    return _admin() != address(0);
  }
}
```

Figure D.2: An Echidna test validating that `address(0)` can never be set.

By running `echidna-test AdminUpgradeTest.sol`, a developer can obtain reasonable assurance that the `AdminUpgradeabilityProxy` contract cannot be forced into an invalid admin state by a malicious user.

The design of AZTEC's Solidity contracts impedes further testing via Echidna:

1. There is no Solidity utility to generate proofs. The AZTEC codebase has high unit test coverage in JavaScript. However, higher confidence in the security of the contracts could be achieved by writing an Echidna test that can assert that every valid proof can be validated, not just the examples in the unit test suite. Because the proofs are generated in JavaScript and not Solidity, we cannot use Echidna to generate several proofs to be tested.
2. There is no valid Solidity ABI to test against. Echidna uses the ABI output of the Solidity compiler to infer the signatures of contract functions. While an ABI specification is provided for each validator contract via the interface contracts, the parameter format does not allow for direct "plug & play" fuzzing. Each validator requires the calldata in a specific format, and since the fuzzer does not have access to this format, it will only produce random invalid bytes. It is possible to write a public ABI that will assert that most proofs do not validate because each calldata specification is static, but this is significantly less interesting.

AZTEC should add property-based testing to their Solidity contract. Unit tests catch cases that have already been considered, but property tests discover edge cases that nobody has thought to consider. With a Solidity proof generator contract, such testing will become more feasible and enable greater assurances that contracts adhere to specifications.

C. Documentation Discrepancies

Throughout the course of the assessment, Trail of Bits discovered issues in the documentation supplied by AZTEC. These issues took the form of both inconsistencies between the documentation and implementation, and actual omissions in the documentation. We recommend that the documentation reflect as accurately as possible the implementation, as this is where the majority of our cryptographic findings arose. This could also potentially be a source of confusion for any users implementing the AZTEC protocol. We document these issues here.

Inconsistencies

- In the AZTEC whitepaper, the JoinSplit algorithm generates random values $(x_0,...,x_n)$, where x_0 is the output of a hash function taken over the input commitments, and the subsequent x_i values are equal to x_0^i. In the implementation, x_0 is computed in the same manner; however, the subsequent x_i values are equal to $H^i(x_0)$, where $H^i(-)$ represents the hash function `H` nested `i` times.
- In the AZTEC whitepaper, pairing checks are used to validate commitments (γ,σ). Specifically, the whitepaper calls for checking that $e(\gamma,t_2) = e(\sigma,g_2)$. However, the implementation actually checks that $e(\gamma,t_2)\cdot e(\sigma^{-1},g_2)=1$.
- In the AZTEC whitepaper, the language used to describe the trusted setup parameters seems to be inconsistent with the actual trusted setup protocol given in the separate documentation. Specifically, the whitepaper claims that `h` and `y` are both chosen uniformly randomly. However, in the actual protocol, `h` depends on `y`, as it is computed by evaluating a polynomial of `y` in the exponent of `g`. It is therefore misleading to say that `h` is chosen uniformly randomly from `G`.

Omissions

- The documentation concerning the trusted setup protocol mentions that each participant's transcripts need to be individually verified. However, they mention no mechanism for each participant to authenticate themselves and their transcript. In the trusted setup implementation, each user generates a random Ethereum account and uses this account to sign their transcript hash as authentication. This information should be included in the documentation with a corresponding security proof.
- As addressed in TOB-AZT-008, verification fails if the random value generated during verification is either zero or one. This information should also be reflected in the documentation.
- The whitepaper presents a JoinSplit protocol and an optimized version of that same protocol. After presenting the optimized protocol, a justification for its security is given, but there's no formal proof of security.

D. Fix Log

Trail of Bits reviewed the fixes proposed by AZTEC for the issues presented in this report. The fixes submitted by AZTEC can be found at the following locations:

- *TOB_AZT_009*: https://github.com/AztecProtocol/AZTEC/pull/317
- *TOB_AZT_010*: https://github.com/AztecProtocol/AZTEC/pull/324
- *TOB_AZT_011*: https://github.com/AztecProtocol/AZTEC/pull/334
- *TOB_AZT_013*: https://github.com/AztecProtocol/AZTEC/pull/331
- *TOB_AZT_015*: https://github.com/AztecProtocol/AZTEC/pull/340
- *TOB_AZT_016*: https://github.com/AztecProtocol/AZTEC/pull/339
- *TOB_AZT_018*: https://github.com/AztecProtocol/AZTEC/pull/301

AZTEC provided fixes for all of our issues except for TOB_AZT_001, TOB_AZT_012, and TOB_AZT_014. Trail of Bits reports no findings for the fixes proposed by AZTEC. As of October 9th, AZTEC has not yet been briefed on the other issues reported by Trail of Bits (TOB_AZT_017, TOB_AZT_019 - TOB_AZT_024). In addition, Trail of Bits presented an update to issue TOB_AZT_015. AZTEC presented a fix for the previous version of TOB_AZT_015, for which we report no findings, but they have not yet been briefed on the updated finding.

#	Title	Severity	Status
1	IERC20 is incompatible with non-standard ERC20 tokens	Medium	Risk Accepted
9	Missing check for address(0) in constructor of AdminUpgradeabilityProxy	Medium	Fixed
10	Missing elliptic curve pairing check in the Swap Validator	High	Fixed
11	Using nested hashes reduces security	Low	Fixed
12	Solidity compiler optimizations can be dangerous	Undetermined	Risk Accepted
13	Replay attack and revocation inversion on confidentialApprove	High	Fixed
14	scalingFactor allows values leading to potentially undesirable behavior	Informational	Risk Accepted

15	Lack of contract existence check on low-level calls will lead to unexpected behavior	Medium	Fixed (previous version of finding)
16	Time locking can overflow	Low	Fixed
18	ZkAssetMintableBase and inherited contracts do not check for proofs supported	Low	Fixed

Detailed Fix Log

Finding 1: IERC20 is incompatible with non-standard ERC20 tokens.
Risk Accepted.

Finding 9: Missing check for `address(0)` in constructor of `AdminUpgradeabilityProxy`.
Fixed.

Finding 10: Missing elliptic curve pairing check in the Swap Validator.
Fixed.

Finding 11: Using nested hashes reduces security.
Fixed.

Finding 12: Solidity compiler optimizations can be dangerous.
Risk Accepted.

Finding 13: Replay attack and revocation inversion on `confidentialApprove.`
Fixed.

Finding 14: `scalingFactor` allows values leading to potentially undesirable behavior.
Risk Accepted.

Finding 15: Lack of contract existence check on low-level calls will lead to unexpected behavior.
Fixed (previous version of finding). At this time, Trail of Bits has not presented its update on this finding to AZTEC, and so AZTEC's fix only applies to the original issue reported in `MultiSigWallet`.

Finding 16: Time locking can overflow.
Fixed.