Veridise. **Auditing Report**

Hardening Blockchain Security with Formal Methods

FOR

Atem Token

Veridise Inc.
October 31, 2023

► **Prepared For:**

Atem Network
https://www.atem.io/

► **Prepared By:**

Jon Stephens

► **Contact Us:** contact@veridise.com

► **Version History:**

Oct. 30, 2023 V1

Contents

1 Executive Summary

On October 30, 2023, Atem Network engaged Veridise to review the security of their Atem Token. The review covered the Solidity source code of the token implementation and a vesting contract that will distribute tokens to a user over time. Veridise conducted the assessment over 1 person-day, with 1 engineers reviewing code over 1 day on the version of the code which we will label V1. After the audit, the fixed version of the code was uploaded to Github and has commit `3ca4e22`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The Atem Token developers provided the source code of the Atem Token contracts for review. The AtemToken inherits the core ERC20 functionality from the OpenZeppelin 4.7.3 ERC20Burnable contract. As a result, they have inherited several safety features such as the `increaseAllowance` and `decreaseAllowance` functions which avoid the potential front-running issues associated with `approve`. In addition to the core ERC20 functionality, the AtemToken allows users to claim airdrops if they either have an approved signature or a merkle tree proof. Once tokens are claimed, a percentage of them may require vesting via the TokenVesting contract. As the name implies, the TokenVesting contract will vest tokens over time. In the version of the contract that was audited, this contract will increase tokens linearly over the duraiton of the vesting period once the cliff has been reached. No external documentation or tests were provided.

Summary of issues detected. The audit uncovered 11 issues, 2 of which are assessed to be of medium severity by the Veridise auditors. Specifically, V-ATN-VUL-001 identifies the possibility for signatures to be replayed on other chains to gain additional funds and V-ATN-VUL-002 identifies a lack of event emits which will make tracking important protocol events difficult. The Veridise auditors also identified several low-severity issues, including the potential for locked funds, a lack of validation on the administrator, and risks associated with centralization.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Project Dashboard 2

Table 2.1: Application Summary.

Name	Version	Type	Platform
Atem Token	V1	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
October 30, 2023	Manual & Tools	1	1 person-day

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	0	0
Medium-Severity Issues	2	2
Low-Severity Issues	5	5
Warning-Severity Issues	3	3
Informational-Severity Issues	1	1
TOTAL	11	11

Table 2.4: Category Breakdown.

Name	Number
Data Validation	3
Logic Error	2
Maintainability	2
Replay Attack	1
Best Practices	1
Locked Funds	1
Centralization	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the AtemToken and the TokenVesting contracts. In our audit, we sought to answer the following questions:

- Can the contract be manipulated such that tokens can be stolen from users?
- Can a user manipulate tokens that are owned by another user?
- Are there risks associated with centralization?
- Are any minting functions properly guarded by access controls?
- Does the token adhere to the behaviors defined in the ERC20 specification?
- Can a user claim more tokens than the specified maximum?
- Can funds be locked in the TokenVesting contract?
- Does the TokenVesting contract properly vest tokens over time?
- Can a user retrieve their funds early from the TokenVesting contract?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancies and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The scope of the audit was limited to `AtemToken_flatten.sol`, which contained the source code of the AtemToken and the TokenVesting contract and the OpenZeppelin 4.7.3 dependencies.

Methodology. Veridise auditors performed a manual audit of the code assisted by both static analyzers and automated testing.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

Table 3.4: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-ATN-VUL-001	Mulit-Chain Replay Attack	Medium	Fixed
V-ATN-VUL-002	Emit in State-Modifying Functions	Medium	Fixed
V-ATN-VUL-003	TokenVesting Contract Locks Native Tokens	Low	Fixed
V-ATN-VUL-004	TokenVesting Contract Accepts All Calls	Low	Fixed
V-ATN-VUL-005	No Validation of Merkle Tree Height	Low	Fixed
V-ATN-VUL-006	No Admin Validation	Low	Fixed
V-ATN-VUL-007	Centralization Risk	Low	Acknowledged
V-ATN-VUL-008	No Validation that a VestingScheduleId isn't in. . .	Warning	Acknowledged
V-ATN-VUL-009	Non-Standard Vesting Cliff	Warning	Fixed
V-ATN-VUL-010	Code Duplication in TokenVesting and AtemToken	Warning	Fixed
V-ATN-VUL-011	Unnecessary Code	Info	Fixed

3.4 Detailed Description of Issues

3.4.1 V-ATN-VUL-001: Mulit-Chain Replay Attack

Severity	Medium	**Commit**	V1
Type	Replay Attack	**Status**	Fixed
File(s)	AtemToken_flatten.sol		
Location(s)	claimWithSignature		
Confirmed Fix At	3ca4e22		

It is currently common practice to deploy contracts on multiple EVM-compatible chains while maintaining the same addresses on each as it simplifies the management of the multi-chain protocol and provides a smoother user-experience. In the case of AtemToken, however, doing so can provide an opportunity for attackers due to how they construct their signature hashes.

When a user submits a claim request with a signature the sender, maximum claim and user type are all hashed as shown below.

```
function claimWithSignature(
    uint256 amount,
    uint256 max_amount,
    uint256 user_type,
    bytes memory signature_
) external nonReentrant() {
    address account = _msgSender();

    require(
            _verifySig(
                keccak256(abi.encodePacked(account, max_amount, user_type)),
                signature_,
                _platformAuthorizeAccount),
            'AtemToken#claimWithSignature: invalid signature of platform authorizer'
        );
    ...
}
```

Figure 3.1: The location in `claimWithSignature` where the initial hash is computed

This is provided to the `_verify` function which uses OpenZeppelin's ECDSA library to construct Ethereum signed message and then to validate that the message was signed by the `_platformAuthorizedAccount`.

Since an Ethereum signed message does not contain a chainid, if AtemToken is deployed on multiple blockchains and `_platformAuthorizedAccount` is maintained, then such signatures can be replayed across chains.

Impact While it appears that signatures are intended to be replayed on a single chain as the signature restricts a user's cumulative claim amount, replaying across chains would effectively allow a user to claim more than the specified max amount.

```
function _verifySig(bytes32 data, bytes memory signature, address account) internal
    pure returns (bool)
{
    return signatureRecover(data, signature) == account;
}

function signatureRecover(bytes32 data, bytes memory signature) public pure returns (
    address) {
    return data
        .toEthSignedMessageHash()
        .recover(signature);
}
```

Figure 3.2: The location in AtemToken where signatures are verified

Recommendation Rather than an Ethereum signed message, consider using EIP712 instead, particularly with a domainSeparator that includes the chainid.

Developer Response We would like to use EIP712, but we do not have time for the front- and back-end to coordinate.

Auditor Response This issue was fixed by adding a chainid that is initialized by an argument in the constructor into the signature hash. With this solution, it is very important that different chainids are used on different chains. The client has been warned of this and they confirmed that they will do so.

3.4.2 V-ATN-VUL-002: Emit in State-Modifying Functions

Severity	Medium	**Commit**	V1
Type	Best Practices	**Status**	Fixed
File(s)	AtemToken_flatten.sol		
Location(s)	AtemToken, TokenVesting		
Confirmed Fix At	3ca4e22		

It is considered best practice to emit an event whenever non-trivial storage modifications are made to a contract. In both the `AtemToken` and `TokenVesting` contracts, however, no events are declared or emitted beside those in the OpenZeppelin library.

```
function setPlatformAuthorizeAccount(address addr) external onlyRole(MINTER_ROLE)
{
    _platformAuthorizeAccount = addr;
}
```

Figure 3.3: Example of an important admin function that does not emit

Impact It is important to emit such events because it (1) makes monitoring the contract for anomalies easier for admins and (2) allows users to monitor the contract for relevant updates (such as vesting schedule changes or revoked vesting).

Recommendation On consequential storage modifications, emit an event with relevant information to provide users and admins with relevant information.

3.4.3 V-ATN-VUL-003: TokenVesting Contract Locks Native Tokens

Severity	Low	**Commit**	`V1`
Type	Locked Funds	**Status**	Fixed
File(s)	`AtemToken_flatten.sol`		
Location(s)	receive, fallback		
Confirmed Fix At	`3ca4e22`		

The solidity language allows developers to define a `receive` function so that they may accept native tokens and perform necessary book keeping. The `TokenVesting` contract defines an empty `receive` function, but does not provide any other functionality to interact with native tokens.

```
/**
 * @dev This function is called for plain Ether transfers, i.e. for every call with
   empty calldata.
 */
receive() external payable {}
```

Figure 3.4: The `receive` function defined in `TokenVesting`

Impact Since a receive function is defined, all native token transfers will be accepted. Since the contract cannot do anything with native tokens though (including rescue them), they will be locked in the contract.

Recommendation Delete the receive function and payable fallback function so that native tokens are rejected.

3.4.4 V-ATN-VUL-004: TokenVesting Contract Accepts All Calls

Severity	Low	**Commit**	`V1`
Type	Logic Error	**Status**	Fixed
File(s)	`AtemToken_flatten.sol`		
Location(s)	fallback		
Confirmed Fix At	`3ca4e22`		

The solidity language allows developers to define a `fallback` function which will be executed when no functions with a given selector match a request. The `TokenVesting` contract defines an empty `fallback` function which will silently accept any function call outside of those defined by the contract.

```
/**
 * @dev Fallback function is executed if none of the other functions match the function
 * identifier or no data was provided with the function call.
 */
fallback() external payable {}
```

Figure 3.5: The `fallback` function defined in `TokenVesting`

Impact The contract will accept any call outside of those defined by the contract, giving the impression that the given function executed as intended. As an example, consider if a user confused the `TokenVesting` address for the `AtemToken` address. If the `transfer` function were called, on `TokenVesting`, it would appear that the transfer was successful since it did not revert (additionally it would be accepted by many `SafeERC20` libraries).

Recommendation Delete the empty `fallback` function so that function calls outside of those defined within the contract revert.

3.4.5 V-ATN-VUL-005: No Validation of Merkle Tree Height

Severity	Low	**Commit**	V1
Type	Data Validation	**Status**	Fixed
File(s)	AtemToken_flatten.sol		
Location(s)	claimWithProof		
Confirmed Fix At	3ca4e22		

The method `claimWithProof` allows a user to provide proof of their membership in a merkle tree to claim funds. During the process of verifying the proof, no validation is performed to check that the proof size matches the expected size of the tree. As an example, `claimWithProof` would accept an empty proof if the provided leaf hashed to the tree root.

```
function _verify(bytes32 leaf, bytes32[] memory proof)
internal view returns (bool)
{
    return MerkleProof.verify(proof, whitelistRoot, leaf);
}
```

Figure 3.6: Location where a merkle tree proof is validated.

Impact This increases the likelihood of collision-based attacks such as second-preimage, dictionary and birthday attacks.

Recommendation Validate the size of the tree when verifying the proof.

3.4.6 V-ATN-VUL-006: No Admin Validation

Severity	Low	**Commit**	V1
Type	Data Validation	**Status**	Fixed
File(s)	AtemToken_flatten.sol		
Location(s)	setPlatformAuthorizeAccount, addVestingSchedule, ...		
Confirmed Fix At	3ca4e22		

Many of the admin functions in the `AtemToken` contract do not validate the inputs provided by the administrator. While it is common for an admin to be a trusted entity, administrator mistakes have led to significant problems in the past. As an example, the new schedules are not validated in `addVestingSchedule` and `modifyVestingSchedule`. In most cases, this will cause claim attempts to revert, such as if `duration` is 0.

```
function addVestingSchedule(
    uint256 tgeRatio,
    uint256 cliff,
    uint256 start,
    uint256 duration,
    uint256 slicePeriodSeconds,
    bool revocable
) external onlyRole(MINTER_ROLE) {
    vestingSchedules.push(VestingScheduleParams({
        tgeRatio: tgeRatio,
        cliff: cliff,
        start: start,
        duration: duration,
        slicePeriodSeconds: slicePeriodSeconds,
        revocable: revocable
    }));
}

function modifyVestingSchedule(
    uint256 index,
    uint256 tgeRatio,
    uint256 cliff,
    uint256 start,
    uint256 duration,
    uint256 slicePeriodSeconds,
    bool revocable
) external onlyRole(MINTER_ROLE) {
    require(index < vestingSchedules.length, "AtemToken#modifyVestingSchedule: invalid index");
    vestingSchedules[index].tgeRatio = tgeRatio;
    vestingSchedules[index].cliff = cliff;
    vestingSchedules[index].start = start;
    vestingSchedules[index].duration = duration;
    vestingSchedules[index].slicePeriodSeconds = slicePeriodSeconds;
    vestingSchedules[index].revocable = revocable;
}
```

Figure 3.7: The definitions of `addVestingSchedule` and `modifyVestingSchedule`

Impact In the cases above, the lack of validation could break functionality in the contract.

Recommendation Validate inputs from admins to prevent potential mistakes.

3.4.7 V-ATN-VUL-007: Centralization Risk

Severity	Low	**Commit**	`V1`
Type	Centralization	**Status**	Acknowledged
File(s)	`AtemToken_flatten.sol`		
Location(s)	N/A		
Confirmed Fix At			

Similar to many projects, the `AtemToken` declares an admin that is given special privileges. In particular, the owner can mint funds, change signing account, change the merkle tree root, add and modify vesting schedules, change the vesting contract, and revoke vesting funds. As these are all particularly sensitive operations, we would encourage the developers to utilize a decentralized governance or multi-sig contract as an EOA introduces a single point of failure.

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, a malicious owner could mint a large number of tokens for themselves, then sell them for a profit, potentially flooding the market.

Recommendation Utilize a decentralized governance or multi-sig contract as the owner of the `AtemToken`.

3.4.8 V-ATN-VUL-008: No Validation that a VestingScheduleId isn't in Use

Severity	Warning	**Commit**	V1
Type	Data Validation	**Status**	Acknowledged
File(s)	AtemToken_flatten.sol		
Location(s)	createVestingSchedule		
Confirmed Fix At			

When a new vesting schedule is created, it is associated with an ID as shown below. The vesting schedule ID is defined to be the hash of the beneficiary and next index to be used in the user's list of vesting schedules. Once the ID is computed, it is then used to store the new vesting schedule without checking to see if the indicated storage slot has already been allocated.

```
function createVestingSchedule(
    ...
) external onlyOwner {
    ...

    bytes32 vestingScheduleId = computeNextVestingScheduleIdForHolder(
        _beneficiary
    );
    uint256 cliff = _start + _cliff;
    vestingSchedules[vestingScheduleId] = VestingSchedule(
        true,
        _beneficiary,
        cliff,
        _start,
        _duration,
        _slicePeriodSeconds,
        _revocable,
        _amount,
        0,
        false
    );

        ...
}
```

Figure 3.8: Location in `createVestingSchedule` where the vesting schedule ID is computed

Impact While collisions are very unlikely, if one were to occur funds would be locked in the contract.

Recommendation Consider requiring that `vestingSchedules[vestingScheduleId]` is not initialized.

3.4.9 V-ATN-VUL-009: Non-Standard Vesting Cliff

Severity	Warning	**Commit**	V1
Type	Logic Error	**Status**	Fixed
File(s)	AtemToken_flatten.sol		
Location(s)	_computeVestedAmount, _computeReleasableAmount		
Confirmed Fix At	3ca4e22		

In the context of vesting, a cliff is typically defined to be a point in time before which nothing is vested and after which all tokens or shares from the start of the vesting period to the cliff are vested at once. In the `TokenVesting` contract, a cliff is essentially defined to be the start of the vesting period. As shown below, once the cliff is reached the amount of tokens vested is 0 since `timeFromCliffEnd` is zero and so `vestedSeconds` is 0. Afterwards, the amount of tokens vested increases linearly until the vesting duration. As a result, the cliff in this case is essentially just the start of the vesting period.

```
function _computeVestedAmount(
    VestingSchedule memory vestingSchedule
) internal view returns (uint256) {
    ...

    // Otherwise, some tokens are releasable.
    else {
        // Compute the number of full vesting periods that have elapsed.
        // uint256 timeFromStart = currentTime - vestingSchedule.start;
        uint256 duration_deduct_cliff = vestingSchedule.duration + vestingSchedule.
    start - vestingSchedule.cliff ;
        uint256 timeFromCliffEnd = currentTime - vestingSchedule.cliff;
        uint256 secondsPerSlice = vestingSchedule.slicePeriodSeconds;
        uint256 vestedSlicePeriods = timeFromCliffEnd / secondsPerSlice;
        uint256 vestedSeconds = vestedSlicePeriods * secondsPerSlice;
        // Compute the amount of tokens that are vested.
        uint256 vestedAmount = (vestingSchedule.amountTotal *
            vestedSeconds) / duration_deduct_cliff;
        // Subtract the amount already released and return.
        return vestedAmount;
    }
}
```

Figure 3.9: The snippet of `_computeVestedAmount` that computes the amount of vested funds

Impact Given that cliff is not typically defined in this way, it could cause confusion with users. Additionally, since the cliff in this case is essentially the start of the vesting period, this is actually complicating the logic since the cliff is included in the duration.

Recommendation Consider either removing the cliff to simplify the logic while maintaining the same functionality or changing the cliff to be in line with the typical definition.

3.4.10 V-ATN-VUL-010: Code Duplication in TokenVesting and AtemToken

Severity	Warning	**Commit**	V1
Type	Maintainability	**Status**	Fixed
File(s)	AtemToken_flatten.sol		
Location(s)	_computeReleasableAmount, _computeVestedAmount		
Confirmed Fix At	3ca4e22		

Two functions in the `TokenVesting` contract define very similar behavior that looks to have been copied and pasted from one location to another.

As an example, consider the `_computeReleasableAmount` function shown below:

```
function _computeReleasableAmount(
    VestingSchedule memory vestingSchedule
) internal view returns (uint256) {
    // Retrieve the current time.
    uint256 currentTime = getCurrentTime();
    // If the current time is before the cliff, no tokens are releasable.
    if ((currentTime < vestingSchedule.cliff) || vestingSchedule.revoked) {
        return 0;
    }
    // If the current time is after the vesting period, all tokens are releasable,
    // minus the amount already released.
    else if (
        currentTime >= vestingSchedule.start + vestingSchedule.duration
    ) {
        return vestingSchedule.amountTotal - vestingSchedule.released;
    }
    // Otherwise, some tokens are releasable.
    else {
        // Compute the number of full vesting periods that have elapsed.
        // uint256 timeFromStart = currentTime - vestingSchedule.start;
        uint256 duration_deduct_cliff = vestingSchedule.duration + vestingSchedule.
    start - vestingSchedule.cliff ;
        uint256 timeFromCliffEnd = currentTime - vestingSchedule.cliff;
        uint256 secondsPerSlice = vestingSchedule.slicePeriodSeconds;
        uint256 vestedSlicePeriods = timeFromCliffEnd / secondsPerSlice;
        uint256 vestedSeconds = vestedSlicePeriods * secondsPerSlice;
        // Compute the amount of tokens that are vested.
        uint256 vestedAmount = (vestingSchedule.amountTotal *
            vestedSeconds) / duration_deduct_cliff;
        // Subtract the amount already released and return.
        return vestedAmount - vestingSchedule.released;
    }
}
```

Figure 3.10: Current definition of the `_computeReleasableAmount` function

It could instead be defined as shown in Figure 3.11 since the definition of `_computeVestedAmount` is exactly the same except it doesn't subtract `vestingSchedule.released` before returning.


```
function _computeReleasableAmount(
    VestingSchedule memory vestingSchedule
) internal view returns (uint256) {
    uint256 vested = _computeVestedAmount(vestingSchedule);
    // Note: when vested == 0, then released == 0 as well
    return vested - vestingSchedule.released;
}
```

Figure 3.11: Simplified definition of `_computeReleasableAmount`

Impact Code duplication can result in maintenance issues in the future since if one location is modified usually all other instances of the same code need to be modified as well. If the code is not modified correctly, then errors are likely to appear.

Recommendation In cases where code is copied, try to reuse the code rather than copy it.

3.4.11 V-ATN-VUL-011: Unnecessary Code

Severity	Info	**Commit**	V1
Type	Maintainability	**Status**	Fixed
File(s)	AtemToken_flatten.sol		
Location(s)	release, releaseAll		
Confirmed Fix At	3ca4e22		

In the `TokenVesting` contract, the address of the beneficiary is cast to a payable address before the transfer. However, an address only needs to be payable if sending native tokens to the recipient, not when sending ERC20 tokens. As a result, the code is unnecessary as the non-payable address can be used.

```
function release(
    bytes32 vestingScheduleId,
    uint256 amount
) public nonReentrant onlyIfVestingScheduleNotRevoked(vestingScheduleId) {
    ...

    address payable beneficiaryPayable = payable(
        vestingSchedule.beneficiary
    );
    vestingSchedulesTotalAmount = vestingSchedulesTotalAmount - amount;
    _token.safeTransfer(beneficiaryPayable, amount);
}
```

Figure 3.12: Location in the `release` function where an address is unnecessarily cast to payable

Recommendation Remove the cast to payable and just use the non-payable address.

