ZKSECURITY

Audit of Aztec's TGE Contract

Date: February 24th, 2025

Introduction

On February 24th, 2025, Aztec engaged zkSecurity to perform an audit of its smart contract related to the Token Generation Event (TGE). The specific code to review was shared via GitHub as a private repository (https://github.com/AztecProtocol/teegeeee at commit `8432c82584731813a2197dd3b715ba2db0dbe3f9`). The audit lasted 3 workdays with 1 consultant.

The code was found to be clear, well documented, and accompanied with thorough tests.

One major finding and a few informational findings were reported to the Aztec team, which are detailed in the following sections.

Scope

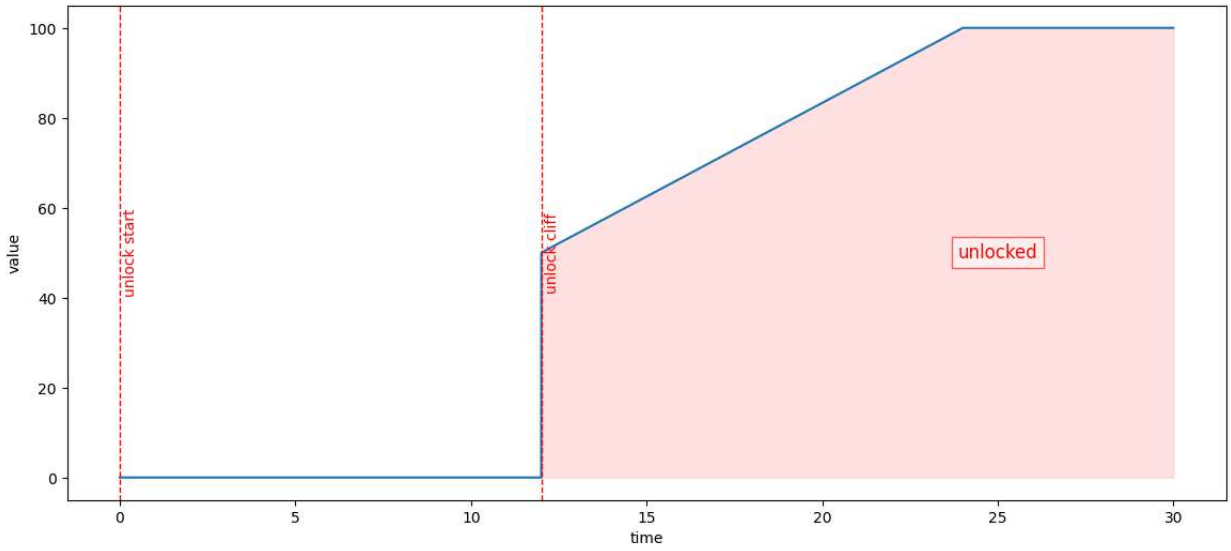
The scope included the Solidity contract in the `teegeeee` repo. At a high level this included:

- `src/token`, which is the AZTEC ERC20 token contract.
- `src/atps`, which is Aztec Token Positions (ATPs), including Milestone Aztec Token Position (MATP) and Linear Aztec Token Position (LATP).
- `src/libraries`, which contains the helper for the schedule lock.
- `src/staker`, which is the staker contract for AZTEC token. Currently, it's just a "no-op" contract and is used for testing.
- `src/ATPFactory.sol`, which is the factory to create new ATPs.
- `src/Registry.sol`, which manages the global unlock schedule, staker implementations, and milestones.

Overview

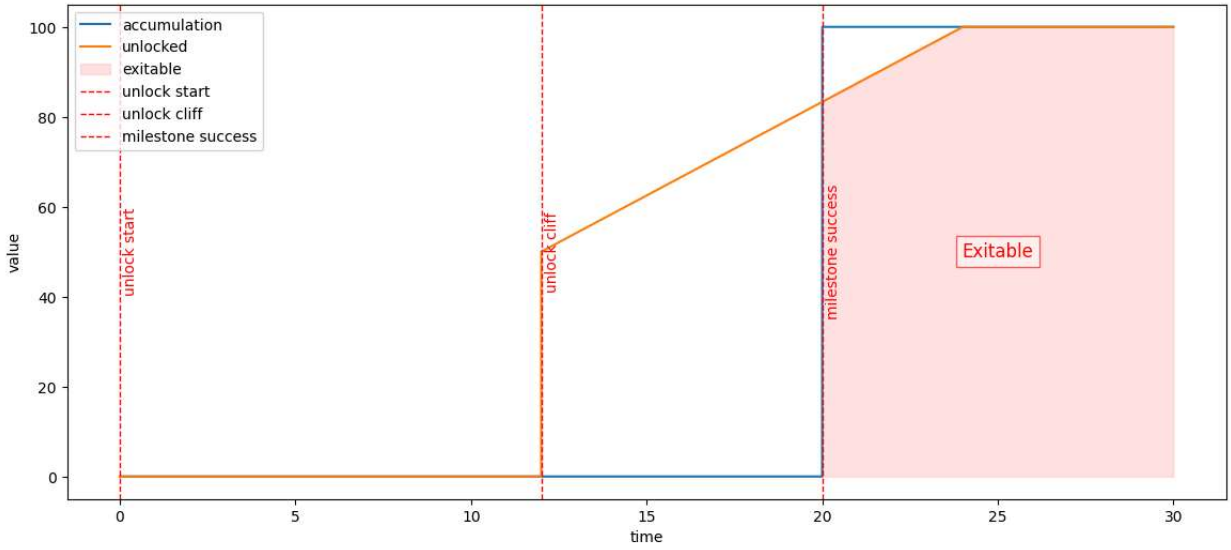
The AZTEC token is an ERC20 token that is mintable by the contract owner. Aztec uses the Aztec Token Positions (ATPs) to distribute the AZTEC token to beneficiaries (e.g., individuals and companies). Each ATP is a standalone contract that holds the tokens to be distributed and specifies the unlock schedule. Typically, the unlock schedule is controlled by two schedules: the global unlock schedule (shared by every ATP) and the local unlock schedule specified by each ATP. At any given time the released token amount is the minimum of the two schedules. For example, if at the 12th month the local schedule releases 20% and the global schedule releases 10%, then the actual release is 10% (min(20%, 10%)). This means that all the ATPs are restricted by the global unlock schedule.

The global unlock schedule is a linear release curve with a cliff. The token is released linearly over time but only claimable after a specific cliff time. The curve is specified by a starting time, cliff time and end time. Below is an example schedule (taken from the docs of the repo) with a 24 month full duration and a 12 month cliff.

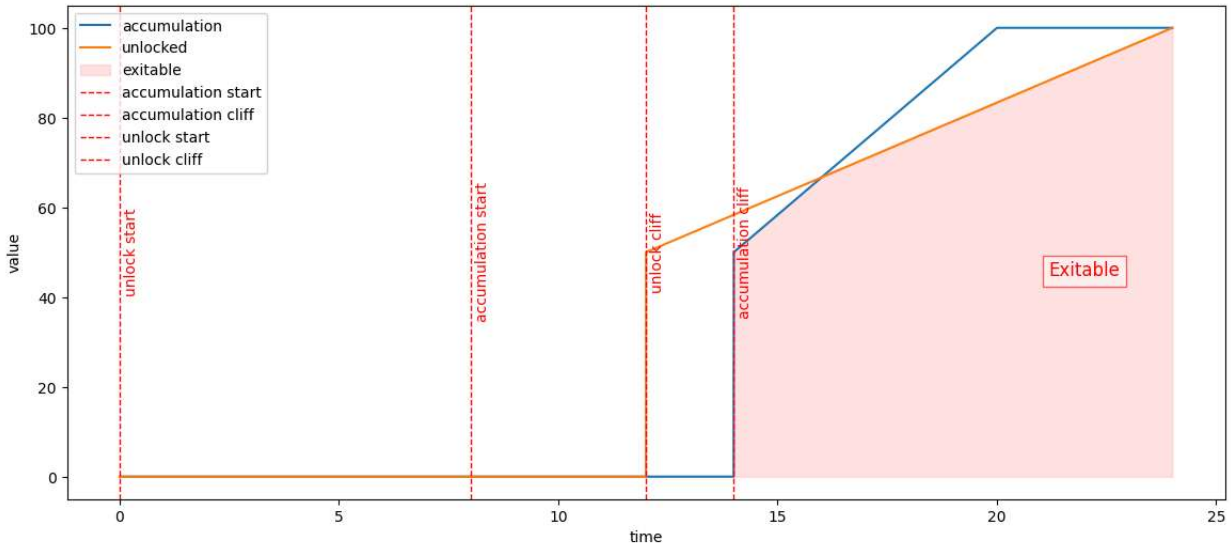


The local unlock schedule has two types: Milestone Aztec Token Position (MATP) and Linear Aztec Token Position (LATP).

The unlock schedule of MATP is specified by a milestone. The token is released only after the milestone (e.g., mainnet launch) is achieved. Before that, all the tokens are locked. Below is an example schedule of the MATP (taken from the docs of the repo). In the example, the tokens are not released after the global unlock cliff but after the milestone success.



The unlock schedule of LATP is a linear release curve with a cliff (just like the global unlock schedule). The resulting unlock amount at a time is the minimum of the global unlock schedule and the local unlock schedule. Below is an example schedule of the MATP (taken from the docs of the repo). In the example, the tokens are released only after the two unlock cliffs are reached.



Staking and Revoking

In the Aztec network, the locked AZTEC token in ATP can participate in staking (to the sequencer). This is achieved by allowing the ATP contract to approve the locked token to the staker contract. The staker can then transfer the token from ATP, perform staking, and transfer the token back when necessary. It is important that the staking operation cannot be used to bypass the unlock schedule. Thus, the ATP can only use the staker contract that is specified by the Aztec-labs. More specifically, each ATP has its own staker contract. The staker contract implementation is whitelisted by the Registry, which is managed by the Aztec-labs.

For some ATP, the tokens can be revoked by a revoker entity if if they have not been accumulated.

For MATP, all the tokens can be revoked by the revoker when the milestone is in `Pending` status (i.e., not `Failed` or `Succeeded`). If the milestone is `Succeeded`, then it can't be revoked and all the tokens will eventually go to the beneficiary (the schedule is still restricted by the global lock). If the milestone is marked as `Failed`, then it means the MATP is fully revoked.

For LATP, there are two cases. It can be set as non-revokable. In this case the local schedule lock is empty and the schedule just follows the global schedule. If it is set as revokable, then it will have a local schedule. The token that is not released according to the local schedule lock is revokable by the revoker. The contract ensures that the revokable portion cannot be transferred to the staker.

Registry

The registry holds the "source of truth" data of the protocol. It consists of three main parts:

- **Global Schedule.** This specifies the global schedule. All ATPs will refer to the registry to get the global schedule. The registry owner can decrease the start time of the global schedule.
- **Staker Implementation.** This contains the implementation address of the staker. When necessary, the registry owner can add new version staker implementations.
- **Milestone.** The registry manages all the milestones. The registry owner can add a new milestone and update the status of it.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
#00	src/atps	The ATP contract can upgrade the Staker to arbitrary address and withdraw the locked token	**High**
#01	src/atps/linear/LATPCore.sol	The beneficiary can claim additional transferred AZTEC tokens before the global lock ends	**Informational**
#02	src/libraries/LockLib.sol	Possible overflow in arithmetic	**Informational**

#00 - The ATP contract can upgrade the Staker to arbitrary address and withdraw the locked token

Severity: High **Location:** src/atps

Description. In the Aztec network, the locked AZTEC tokens in ATP can participate in staking. This is achieved by allowing the ATP contract to approve the locked token to the staker contract. It is crucial that the staking operation cannot be used to bypass the unlock schedule. In the contract, the ATP should be restricted to only use the staker contract that is whitelisted by the Registry. Unfortunately, due to an oversight, the ATP contract can use the `upgradeStaker` function to bypass the restriction and upgrade the staker contract to an arbitrary contract.

```
function upgradeStaker(StakerVersion _version, bytes memory _initdata)
    external
    override(IMATPCore)
    onlyBeneficiary
{
    address impl = REGISTRY.getMilestoneStakerImplementation(_version);
    UUPSUpgradeable(address(staker)).upgradeToAndCall(impl, _initdata);

    emit StakerUpgraded(_version);
}
```

In the `upgradeStaker` function above, the caller specifies the staker `_version` and `_initdata`. The staker implementation is then fetched from the global Registry. Then it will perform a UUPS upgrade for the staker and call the function specified in the `_initdata`. The `_initdata` is intended for initialization operations. However, there is no restriction on the `_initdata`, allowing the ATP to call arbitrary functions with arbitrary parameters during the upgrade. This includes calling the `upgradeToAndCall` function again.

To perform the attack, the ATP contract will encode the `upgradeToAndCall` function in the `_initdata`. Calling the `upgradeStaker` function then leads to two upgrades. The first upgrade has the implementation address checked. However, the second upgrade is embedded in `upgradeToAndCall` and is not checked. As a result, the ATP can upgrade the staker to an arbitrary implementation.

Below is a PoC of the attack:

```
// Add this test in test/btt/atps/milestone/upgradeStaker/upgradeStaker.t.sol

function test_arbitraryUpgrade() external {
    address initialStaker =
registry.getMilestoneStakerImplementation(StakerVersion.wrap(0));
    FakeMilestoneStaker badStaker = new FakeMilestoneStaker(staking);
    // encode the UUPSUpgradeable.upgradeToAndCall function with badStaker in
the _initdata
    atp.upgradeStaker(StakerVersion.wrap(0),
abi.encodeCall(UUPSUpgradeable.upgradeToAndCall, (address(badStaker), "")));

    assertEq(atp.getStaker().getImplementation(), address(badStaker));
    assertNotEq(address(badStaker), initialStaker);
}
```

Impact. The ATP beneficiary can exploit this vulnerability to upgrade the staker to malicious implementation and then withdraw the locked token from the staker.

Recommendation. It is implicit that the ATP contract can call any function of the staker contract though the `upgradeStaker` function. To mitigate this, it is recommended to add a restriction to the `_initdata` in the function. For example, only allow calling the `initialize` function in the `_initdata`.

#01 - The beneficiary can claim additional transferred AZTEC tokens before the global lock ends

Severity: Informational **Location:** src/atps/linear/LATPCore.sol

Description. Besides the token to be distributed, the ATP contract may also hold additional transferred tokens. For example, users may transfer their own AZTEC tokens to the ATP contract directly (by mistake). The contract supports that the beneficiary can claim these tokens only after the global lock ends. As described in the documentation "Any additional tokens transferred to the ATP after its creation cannot be retrieved until after the global lock ends.". However, it turns out that, in the LATP contract, the beneficiary can still retrieve the additional transferred AZTEC tokens before the global lock ends.

In LATP, the claimable amount is calculated as follows:

```
function getClaimable() public view override(IATPCore) returns (uint256) {
    Lock memory globalLock = getGlobalLock();
    uint256 unlocked = globalLock.hasEnded(block.timestamp)
        ? type(uint256).max
        : (globalLock.unlockedAt(block.timestamp) - claimed);

    return Math.min(TOKEN.balanceOf(address(this)) - getRevokableAmount(),
unlocked);
}
```

When the accumulated amount is lower than the global unlock amount, the claimable amount is calculated as `TOKEN.balanceOf(address(this)) - getRevokableAmount()`. This calculation includes additional transferred AZTEC tokens.

Impact. This behavior does not cause critical issues but is inconsistent with the documentation.

Recommendation. It is recommended to align the documentation with the contract behavior.

#02 - Possible overflow in arithmetic

Severity: Informational **Location:** src/libraries/LockLib.sol

Description. The LockLib contains helper functions to handle the schedule lock. The `createLock` function creates a lock based on the provided parameters:

```
function createLock(LockParams memory _params, uint256 _allocation) internal
pure returns (Lock memory) {
    LockLib.assertValid(_params);
    return Lock({
        startTime: _params.startTime,
        cliff: _params.startTime + _params.cliffDuration,
        endTime: _params.startTime + _params.lockDuration,
        allocation: _allocation
    });
}
```

The cliff time and end time are calculated as the sum of the start time and the duration. The sum could overflow the range of uint256 if the time and duration are large numbers.

Additionally, in the `claim` function of ATP contract, the total claimed amount is updated by adding the current claimed amount. This could cause overflow if the amount is very large and the beneficiary cyclically claims and transfers tokens (since additional transferred token can be claimed).

```
function claim() external override(IMATPCore) onlyBeneficiary returns (uint256)
{
    uint256 amount = getClaimable();
    require(amount > 0, NoClaimable());

    claimed += amount;

    TOKEN.safeTransfer(msg.sender, amount);

    emit Claimed(amount);
    return amount;
}
```

Impact. This is unlikely to cause issues in practice, as such overflows are improbable in real-world scenarios.

Recommendation. It is recommended to use SafeMath for arithmetic operations to prevent potential overflow and underflow.