Glif Filecoin InfinityPool

Date	April 2023
Auditors	Chingiz Mardanov, Sergii Kravchenko

1 Executive Summary

This report presents the results of our engagement with the GLIF team to review a new Filecoin leasing protocol for Filecoin Storage Providers called GLIF Pools. The goal of the Protocol is to make possible the deployment of multiple Filecoin leasing pools, allowing a single Agent (representing a Filecoin Storage Provider) to lease FIL for use as pledge collateral on the Filecoin blockchain. This audit focused on the underlying Protocol code, as well as the Infinity Pool implementation as the first leasing pool built on the Protocol.

The review was conducted over three weeks, from **April 10th, 2023** to **April 28th, 2023**, by **Chingiz Mardanov** and **Sergii Kravchenko**. A total of **5 person-weeks** were spent. The fixes were reviewed over **3 days**, from **May 3d, 2023** to **May 5th, 2023**.

Prior to this review, a 10 person-weeks, informal protocol review occurred, after which the code was significantly changed. We are happy to mention that the code size, quality and business logic were drastically improved, making it easier to understand and audit the codebase. While this is the case, we still have identified several major issues that needed to be addressed. We also would like to mention that given the short time alloted for the audit and several trust assumptions the Protocol has to be used cautiously.

The GLIF Pools protocol has a large potential feature set and surface area, so the following assumptions were made about the state of the system in order to conduct an audit in a reasonable amount of time:

- The system contains only 1 leasing pool.
- The Agent Data Oracle is controlled solely by the GLIF team.
- All owner and operator keys with control over important parts of the Protocol are multisig wallets controlled by the GLIF team.
- The Filecoin precompiles used throughout the Protocol to interact with Filecoin's built-in Minor Actor have already been audited.

Any changes to the protocol code, such as upgrades, adding off-ramp or adding additional pools, must undergo additional audits.

The security of this Protocol relies heavily on the centralization nature of its current state and any decrease in the centralization must be carefully evaluated and audited. Users must understand that the GLIF Pools Protocol is centralized and controlled by the GLIF team. We also talk about this in Trust Model section. All the issues were addressed by the GLIF team, the details of which can be found in the Findings section.

2 Scope

Our review focused on the commit hash `bf28b412e0e13c87b1dfdcf4cb6ee2707bbe44f8`. The list of files in scope can be found in the Appendix. The fixes were presented on the commit hash `070dae7820c9299eb610f3300b0ae5e35b139913`.

2.1 Objectives

Together with the **GLIF** team, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

3.1 Actors

In the current system we can identify a few key actors. In order to simplify things we will do that from two different levels: external and internal relative the protocol.

On external level we have 3 main actors:

- **Depositors** - users who are willing to provide capital depending on the terms of the specific pool. From their perspective they are interacting with an ERC4626 vault that will have withdrawals done via a queuing system.
- **Filecoin Storage Providers** - are the once that will borrow the capital provided by the depositors. Those funds can either be withdrawn, when correct computerization checks are passed or used to further expand the storage provision operation.

- **GLIF Protocol** - acts as a place where depositors and borrowers can be matched based on the deals they would like to support.
- **Oracle** - is used to get the miners information of the storage provider is initially controlled by the GLIF team. Values that are passed using this oracle are crucial to the correct operation of the protocol.

On the internal level things are a bit more complex. Now we will outline the main participants that build up the main protocol.

InfinityPool - is an actor where the main actions of the protocol would happen. This pools is where depositors can deposit their funds wFIL and FIL in the current iteration and Agents can later borrow those funds.

Agent - is the contract that represents the borrower in the context of the protocol. Agent will obtain control over all of the storage provider's miners. Transferring the ownership over the miners to the Agent contract is what prevents the borrower from running away with the borrowed funds. Agent can be though of as borrowers bank account and only funds that are not used as collateral can be withdrawn from that account.

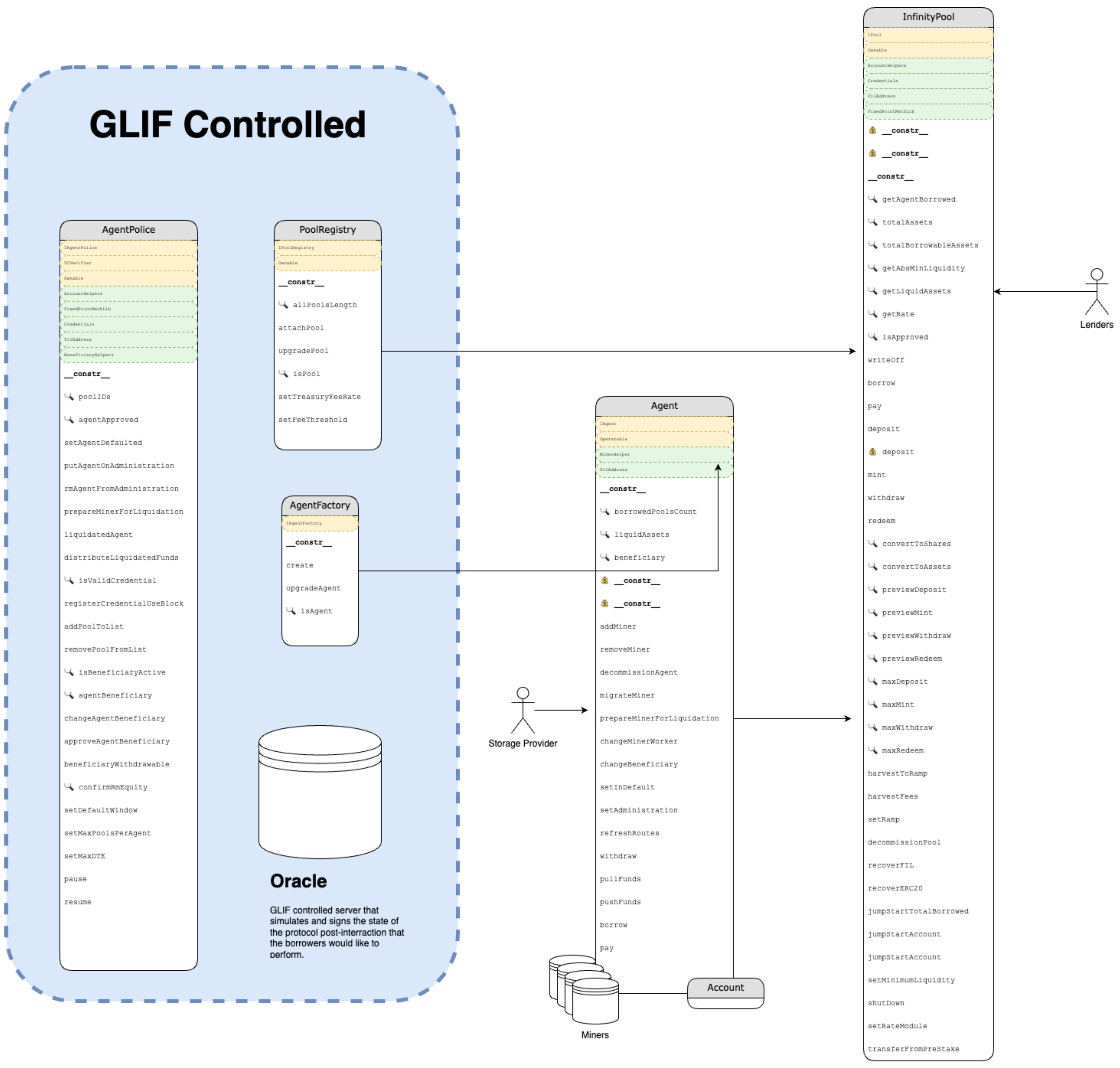
AgentPolice - is a contract that will be controlled by the GLIF team from launch. AgentPolice verifies actions that are taking place on chain as well as oversee processes such as putting an Agent in administration or liquidating and Agent in default.

3.2 Trust Model

After thoroughly reviewing the codebase we would like to highlight that it heavily relies on a strong trust assumption that the development team will not only act with integrity and good faith but also exercise a high level of caution when making any future code changes. While we understand that trust is a fundamental aspect of any collaborative project, it is important to acknowledge that this trust assumption presents a significant risk to the security and reliability of the system. Such trust assumption include but are not limited to:

- Upgradeability of the majority of the contract such as `PoolRegistry`, `MinerRegistry`, `Agent` e.t.c.
- Since upgradeability is done in a rather complex manner of redeploying contracts with previously non-empty state any upgrades should be done extremely carefully after an extensive testing and verification.
- For proper operation all actions that borrowers can take rely on the data submitted by a centralized oracle operated by the GLIF team. If that oracle starts reporting incorrect values the entire protocol security will be jeopardized.
- Entire concept of administration as well as the fact that AgentPolice controlled by the GLIF team, means that there are several ways in which the GLIF team could hijack the users funds. This applies to both borrowers and depositors.

4 System Overview



- **PoolRegistry** - A contract that keeps track of all the leasing pools. At launch there will be only one pool called InfinityPool. We operated under assumption that given code base will only work with one leasing pool.
- **InfinityPool** - Main contract where most of the borrowing and leasing logic is contained.
- **PoolToken:Share** - Contract that will represent the share of the depositor in a particular pool.
- **PoolToken:iou** - Token that represents the debt of the pool in front of the depositor. You can also think of this as a ticket in a withdraw queue.

- **Agent** - A contract that will control the storage providers miners or any funds that can not be withdrawn. All the actions that the storage providers would like to perform will be done via the Agent contract.
- **Account** - A new struct that is created every time a lessor's Agent opens a borrow position in a new pool.
- **AgentPolice** - Contract controlled by the GLIF team that allows them to take control over agents and perform liquidations.
- **Centralized Oracle** - A server that reports the signed on chain data to the Agent. This information is required to make sure that Agent can not enter bad state such as over-collateralized state for example.

5 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 InfinityPool contract authorization bypass attack Critical ✓ Fixed

Resolution

Addressed by not allowing the `vc.subject` to be zero.

Description

An attacker could create their own credential and set the Agent ID to `0`, which would bypass the `subjectIsAgentCaller` modifier. The attacker could use this attack to borrow funds from the pool, draining any available liquidity. For example, only an `Agent` should be able to borrow funds from the pool and call the `borrow` function:

code/src/Pool/InfinityPool.sol:L302-L325

```
function borrow(VerifiableCredential memory vc) external isOpen subjectIsAgentCaller(vc) {
    // 1e18 => 1 FIL, can't borrow less than 1 FIL
    if (vc.value < WAD) revert InvalidParams();
    // can't borrow more than the pool has
    if (totalBorrowableAssets() < vc.value) revert InsufficientLiquidity();
    Account memory account = _getAccount(vc.subject);
    // fresh account, set start epoch and epochsPaid to beginning of current window
    if (account.principal == 0) {
        uint256 currentEpoch = block.number;
        account.startEpoch = currentEpoch;
        account.epochsPaid = currentEpoch;
        GetRoute.agentPolice(router).addPoolToList(vc.subject, id);
    }

    account.principal += vc.value;
    account.save(router, vc.subject, id);

    totalBorrowed += vc.value;

    emit Borrow(vc.subject, vc.value);

    // interact - here `msg.sender` must be the Agent bc of the `subjectIsAgentCaller` modifier
    asset.transfer(msg.sender, vc.value);
}
```

The following modifier checks that the caller is an `Agent`:

code/src/Pool/InfinityPool.sol:L96-L101

```
modifier subjectIsAgentCaller(VerifiableCredential memory vc) {
    if (
        GetRoute.agentFactory(router).agents(msg.sender) != vc.subject
    ) revert Unauthorized();
    _;
}
```

But if the caller is not an `Agent`, the `GetRoute.agentFactory(router).agents(msg.sender)` will return `0`. And if the `vc.subject` is also zero, the check will be successful with any `msg.sender`. The attacker can also pass an arbitrary `vc.value` as the parameter and steal all the funds from the pool.

Recommendation

Ensure only an `Agent` can call `borrow` and pass the `subjectIsAgentCaller` modifier.

5.2 Agent Data Oracle signed credential front-running attack Major ✓ Fixed

Resolution

Mitigated by allowing only the `Agent` to request credentials.

Description

For almost every action as an `Agent`, the owner of the `Agent` is supposed to request `SignedCredential` data that contains all the relevant current info about the "off-chain" state of the `Agent`. New credentials can only be requested when the old one for this `Agent` is used or expired. Anyone can request these credentials, containing all the data about the call. So if the attacker consistently requests the credentials with the function and parameters that the actual `Agent` wouldn't want to call, the `Agent` won't be able to generate the credentials that are needed.

Recommendation

Ensure an `Agent` can always have new credentials that are needed. One solution would be to allow only an Agent's owner to request the credentials. The problem is that the beneficiary is also supposed to do that, but the beneficiary may also be a contract.

5.3 Wrong accounting for `totalBorrowed` in the `InfinityPool.writeOff` function Major ✓ Fixed

Resolution

Fixed.

Description

Here is a part of the `InfinityPool.writeOff` function:

code/src/Pool/InfinityPool.sol:L271-L287

```
// transfer the assets into the pool
// whatever we couldn't pay back
uint256 lostAmt = principalOwed > recoveredFunds ? principalOwed - recoveredFunds : 0;

uint256 totalOwed = interestPaid + principalOwed;

asset.transferFrom(
  msg.sender,
  address(this),
  totalOwed > recoveredFunds ? recoveredFunds : totalOwed
);
// write off only what we lost
totalBorrowed -= lostAmt;
// set the account with the funds the pool lost
account.principal = lostAmt;

account.save(router, agentID, id);
```

The `totalBorrowed` is decreased by the `lostAmt` value. Instead, it should be decreased by the original `account.principal` value to acknowledge the loss.

5.4 Wrong accounting for `totalBorrowed` in the `InfinityPool.pay` function Major ✓ Fixed

Resolution

Addressed as recommended in two pull rquests: 1, 2.

Description

If the Agent pays more than the current interest debt, the remaining payment will be accounted as repayment of the principal debt:

code/src/Pool/InfinityPool.sol:L382-L401

```
// pay interest and principal
principalPaid = vc.value - interestOwed;
// the fee basis only applies to the interest payment
feeBasis = interestOwed;
// protect against underflow
totalBorrowed -= (principalPaid > totalBorrowed) ? 0 : principalPaid;
// fully paid off
if (principalPaid >= account.principal) {
    // remove the account from the pool's list of accounts
    GetRoute.agentPolice(router).removePoolFromList(vc.subject, id);
    // return the amount of funds overpaid
    refund = principalPaid - account.principal;
    // reset the account
    account.reset();
} else {
    // interest and partial principal payment
    account.principal -= principalPaid;
    // move the `epochsPaid` cursor to mark the account as "current"
    account.epochsPaid = block.number;
}
```

Let's focus on the `totalBorrowed` changes:

code/src/Pool/InfinityPool.sol:L387

```
totalBorrowed -= (principalPaid > totalBorrowed) ? 0 : principalPaid;
```

This value is supposed to be decreased by the principal that is repaid. So there are 2 mistakes in the calculation:

- Should be `totalBorrowed` instead of `0`.
- The `principalPaid` cannot be larger than the `account.principal` in that calculation.

5.5 The `beneficiaryWithdrawable` function can be called by anyone Major ✓ Fixed

Resolution

Fixed by removing beneficiary logic completely.

Description

The `beneficiaryWithdrawable` function is supposed to be called by the Agent when a beneficiary is trying to withdraw funds:

code/src/Agent/AgentPolice.sol:L320-L341

```
function beneficiaryWithdrawable(
  address recipient,
  address sender,
  uint256 agentID,
  uint256 proposedAmount
) external returns (
  uint256 amount
) {
  AgentBeneficiary memory beneficiary = _agentBeneficiaries[agentID];
  address benneficiaryAddress = beneficiary.active.beneficiary;
  // If the sender is not the owner of the Agent or the beneficiary, revert
  if(
    !(benneficiaryAddress == sender || (IAuth(msg.sender).owner() == sender && recipient == benneficiaryAddress) )) {
      revert Unauthorized();
    }
  (
    beneficiary,
    amount
  ) = beneficiary.withdraw(proposedAmount);
  // update the beneficiary in storage
  _agentBeneficiaries[agentID] = beneficiary;
}
```

This function reduces the quota that is supposed to be transferred during the `withdraw` call:

code/src/Agent/Agent.sol:L343-L352

```
  sendAmount = agentPolice.beneficiaryWithdrawable(receiver, msg.sender, id, sendAmount);
}
else if (msg.sender != owner()) {
  revert Unauthorized();
}

// unwrap any wfil needed to withdraw
_poolFundsInFIL(sendAmount);
// transfer funds
payable(receiver).sendValue(sendAmount);
```

The issue is that anyone can call this function directly, and the quota will be reduced without funds being transferred.

Recommendation

Ensure only the Agent can call this function.

5.6 An `Agent` can borrow even with existing debt in interest payments Medium ✓ Fixed

Resolution

Mitigated by adding a limit to the remaining interest debt when borrowing. So an agent should have an interest debt that is no larger than 1 day.

Description

To borrow funds, an `Agent` has to call the `borrow` function of the pool:

code/src/Pool/InfinityPool.sol:L302-L325

```
function borrow(VerifiableCredential memory vc) external isOpen subjectIsAgentCaller(vc) {
    // 1e18 => 1 FIL, can't borrow less than 1 FIL
    if (vc.value < WAD) revert InvalidParams();
    // can't borrow more than the pool has
    if (totalBorrowableAssets() < vc.value) revert InsufficientLiquidity();
    Account memory account = _getAccount(vc.subject);
    // fresh account, set start epoch and epochsPaid to beginning of current window
    if (account.principal == 0) {
        uint256 currentEpoch = block.number;
        account.startEpoch = currentEpoch;
        account.epochsPaid = currentEpoch;
        GetRoute.agentPolice(router).addPoolToList(vc.subject, id);
    }

    account.principal += vc.value;
    account.save(router, vc.subject, id);

    totalBorrowed += vc.value;

    emit Borrow(vc.subject, vc.value);

    // interact - here `msg.sender` must be the Agent bc of the `subjectIsAgentCaller` modifier
    asset.transfer(msg.sender, vc.value);
}
```

Let's assume that the `Agent` already had some funds borrowed. During this function execution, the current debt status is not checked. The principal debt increases after borrowing, but `account.epochsPaid` remains the same. So the pending debt will instantly increase as if the borrowing happened on `account.epochsPaid`.

Recommendation

Ensure the debt is paid when borrowing more funds.

5.7 The `AgentPolice.distributeLiquidatedFunds()` function can have undistributed residual funds Medium ✓ Fixed

Resolution

Mitigated by returning the excess funds in `wFil` to the `Agent`'s owner. The only trick here is that the `Agent`'s owner should be able to manage these funds.

Description

When an Agent is liquidated, the liquidator (owner of the protocol) is supposed to try to redeem as many funds as possible and re-distribute them to the pools:

code/src/Agent/AgentPolice.sol:L185-L191

```
function distributeLiquidatedFunds(uint256 agentID, uint256 amount) external {
  if (!liquidated[agentID]) revert Unauthorized();

  // transfer the assets into the pool
  GetRoute.wFIL(router).transferFrom(msg.sender, address(this), amount);
  _writeOffPools(agentID, amount);
}
```

The problem is that in the pool, it's accounted that the amount of funds can be larger than the debt. In that case, the pool won't transfer more funds than the pool needs:

code/src/Pool/InfinityPool.sol:L275-L289

```
uint256 totalOwed = interestPaid + principalOwed;

asset.transferFrom(
  msg.sender,
  address(this),
  totalOwed > recoveredFunds ? recoveredFunds : totalOwed
);
// write off only what we lost
totalBorrowed -= lostAmt;
// set the account with the funds the pool lost
account.principal = lostAmt;

account.save(router, agentID, id);

emit WriteOff(agentID, recoveredFunds, lostAmt, interestPaid);
```

If that happens, the remaining funds will be stuck in the `AgentPolice` contract.

Recommendation

Return the residual funds to the Agent's owner or process them in some way so they are not lost.

5.8 An `Agent` can be upgraded even if there is no new implementation Medium ✓ Fixed

Resolution

Mitigated by introducing a new version control mechanism. This solution also adds centralized power. The owner can create a new deployer with an arbitrary (even lower) version number, while agents can only upgrade to a higher version. Also, agents are forced to upgrade to a new version in another pull request.

Description

Agents can be upgraded to a new implementation, and only the Agent's owner can call the upgrade function:

code/src/Agent/AgentFactory.sol:L51-L72

```
function upgradeAgent(
  address agent
) external returns (address newAgent) {
  IAgent oldAgent = IAgent(agent);
  address owner = IAuth(address(oldAgent)).owner();
  uint256 agentId = agents[agent];
  // only the Agent's owner can upgrade, and only a registered agent can be upgraded
  if (owner != msg.sender || agentId == 0) revert Unauthorized();
  // deploy a new instance of Agent with the same ID and auth
  newAgent = GetRoute.agentDeployer(router).deploy(
    router,
    agentId,
    owner,
    IAuth(address(oldAgent)).operator()
  );
  // Register the new agent and unregister the old agent
  agents[newAgent] = agentId;
  // transfer funds from old agent to new agent and mark old agent as decommissioning
  oldAgent.decommissionAgent(newAgent);
  // delete the old agent from the registry
  agents[agent] = 0;
}
```

The issue is that the owner can trigger the upgrade even if no new implementation exists. Multiple possible problems derive from it.

- Upgrading to the current implementation of the Agent will break the logic because the current version is not calling the `migrateMiner` function, so all the miners will stay with the old Agent, and their funds will be lost.
- The owner can accidentally trigger multiple upgrades simultaneously, leading to a loss of funds (https://github.com/ConsenSysDiligence/glif-audit-2023-04/issues/2).

The owner also has no control over the new version of the Agent. To increase decentralization, it's better to pass the deployer's address as a parameter additionally.

Recommendation

Ensure the upgrades can only happen when there is a new version of an Agent, and the owner controls this version.

5.9 Potential re-entrancy issues when upgrading the contracts Minor ✓ Fixed

Resolution

The issue is mitigated by removing the old agent before the potential re-entrancy.

Description

The protocol doesn't have any built-in re-entrancy protection mechanisms. That mainly explains by using the `wFIL` token, which is not supposed to give that opportunity. And also by carefully using `FIL` transfers.

However, there are some places in the code where things may go wrong in the future. For example, when upgrading an `Agent`:

code/src/Agent/AgentFactory.sol:L51-L72

```
function upgradeAgent(
  address agent
) external returns (address newAgent) {
  IAgent oldAgent = IAgent(agent);
  address owner = IAuth(address(oldAgent)).owner();
  uint256 agentId = agents[agent];
  // only the Agent's owner can upgrade, and only a registered agent can be upgraded
  if (owner != msg.sender || agentId == 0) revert Unauthorized();
  // deploy a new instance of Agent with the same ID and auth
  newAgent = GetRoute.agentDeployer(router).deploy(
    router,
    agentId,
    owner,
    IAuth(address(oldAgent)).operator()
  );
  // Register the new agent and unregister the old agent
  agents[newAgent] = agentId;
  // transfer funds from old agent to new agent and mark old agent as decommissioning
  oldAgent.decommissionAgent(newAgent);
  // delete the old agent from the registry
  agents[agent] = 0;
}
```

Here, we see the `oldAgent.decommissionAgent(newAgent);` call happens before the `oldAgent` is deleted. Inside this function, we see:

code/src/Agent/Agent.sol:L200-L212

```
function decommissionAgent(address _newAgent) external {
  // only the agent factory can decommission an agent
  AuthController.onlyAgentFactory(router, msg.sender);
  // if the newAgent has a mismatching ID, revert
  if(IAgent(_newAgent).id() != id) revert Unauthorized();
  // set the newAgent in storage, which marks the upgrade process as starting
  newAgent = _newAgent;
  uint256 _liquidAssets = liquidAssets();
  // Withdraw all liquid funds from the Agent to the newAgent
  _poolFundsInFIL(_liquidAssets);
  // transfer funds to new agent
  payable(_newAgent).sendValue(_liquidAssets);
}
```

Here, the FIL is transferred to a new contract which is currently unimplemented and unknown. Potentially, the fallback function of this contract could trigger a re-entrancy attack. If that's the case, during the execution of this function, there will be two contracts that are active agents with the same ID, and the attacker can try to use that maliciously.

Recommendation

Be very cautious with further implementations of agents and pools. Also, consider using reentrancy protection in public functions.

5.10 InfinityPool is subject to a donation with inflation attack if emtied. Minor ✓ Fixed

Resolution

this issue will not be fixed in the current version of the contracts since some of the shares were already minted. The next iteration of the pool will have a more generic fix to this issue.

Description

Since `InfinityPool` is an implementation of the ERC4626 vault, it is too susceptible to inflation attacks. An attacker could front-run the first deposit and inflate the share price to an extent where the following deposit will be less than the value of 1 wei of share resulting in 0 shares minted. The attacker could conduct the inflation by means of self-destructing of another contract. In the case of GLIF this attack is less likely on the first pool since GLIF team accepts predeposits so some amount of shares was already minted. We do suggest fixing this issue before the next pool is deployed and no pre-stake is generated.

Examples

code/src/Pool/InfinityPool.sol:L491-L516

```
/*//////////////////////////////////////////////////////////////
                        4626 LOGIC
//////////////////////////////////////////////////////////////*/

/**
 * @dev Converts `assets` to shares
 * @param assets The amount of assets to convert
 * @return shares - The amount of shares converted from assets
 */
function convertToShares(uint256 assets) public view returns (uint256) {
    uint256 supply = liquidStakingToken.totalSupply(); // Saves an extra SLOAD if totalSupply is non-zero.

    return supply == 0 ? assets : assets * supply / totalAssets();
}

/**
 * @dev Converts `shares` to assets
 * @param shares The amount of shares to convert
 * @return assets - The amount of assets converted from shares
 */
function convertToAssets(uint256 shares) public view returns (uint256) {
    uint256 supply = liquidStakingToken.totalSupply(); // Saves an extra SLOAD if totalSupply is non-zero.

    return supply == 0 ? shares : shares * totalAssets() / supply;
}
```

Recommendation

Since the pool does not need to accept donations, the easiest way to handle this case is to use virtual price, where the balance of the contract is duplicated in a separate variable.

5.11 `MaxWithdraw` should potentially account for the funds available in the ramp. Minor ✓ Fixed

Resolution

Partially fixed in https://github.com/glif-confidential/pools/issues/462 but the ramp balance is still not accounted for.

Description

Since `InfinityPool` is ERC4626 it should also support the `MaxWithdraw` method. According to the EIP it should include any withdrawal limitation that the participant could encounter. At the moment the `MaxWithdraw` function returns the maximum amount of IOU tokens rather than WFIL. Since IOU token is not the `asset` token of the vault, this behavior is not ideal.

Examples

code/src/Pool/InfinityPool.sol:L569-L571

```
function maxWithdraw(address owner) public view returns (uint256) {
    return convertToAssets(liquidStakingToken.balanceOf(owner));
}
```

Recommendation

We suggest considering returning the maximum amount of WFIL withdrawal which should account for Ramp balance.

5.12 The upgradeability of MinerRegistry, AgentPolice, and Agent is overcomplicated and has a hight chance of errors. Minor Acknowledged

Description

During the engagement, we have identified a few places that signify that the `Agent`, `MinerRegistry` and `AgentPolice` can be upgraded, for example:

- Ability to migrate the miner from one version of the Agent to another inside the `migrateMiner`.
- Ability to `refreshRoutes` that would update the `AgentPolice` and `MinerRegistry` addresses for a given Agent.
- Ability to `decommission` pool. We believe that this functionality is present it is not very well thought through. For example, both `MinerRegistry` and `AgentPolice` are not upgradable but have mappings inside of them.

code/src/Agent/AgentPolice.sol:L51-L60

```
mapping(uint256 => bool) public liquidated;

/// @notice `_poolIDs` maps agentID to the pools they have actively borrowed from
mapping(uint256 => uint256[]) private _poolIDs;

/// @notice `_credentialUseBlock` maps signature bytes to when a credential was used
mapping(bytes32 => uint256) private _credentialUseBlock;

/// @notice `_agentBeneficiaries` maps an Agent ID to its Beneficiary struct
mapping(uint256 => AgentBeneficiary) private _agentBeneficiaries;
```

code/src/Agent/MinerRegistry.sol:L18-L20

```
mapping(bytes32 => bool) private _minerRegistered;

mapping(uint256 => uint64[]) private _minersByAgent;
```

That means that any time these contracts would need to be upgraded, the contents of those mappings will need to be somehow recreated in the new contract. That is not trivial since it is not easy to obtain all values of a mapping. This will also require an additional protocol-controlled setter ala kickstart mapping functions that are not ideal.

In the case of `Agent` if the contract was upgradable there would be no need for a process of migrating miners that can be tedious and opens possibilities for errors. Since protocol has a lot of centralization and trust assumptions already, having upgradability will not contribute to it a lot.

We also believe that during the upgrade of the pool, the PoolToken will stay the same in the new pool. That means that the minting and burning permissions of the share tokens have to be carefully updated or checked in a manner that does not require the address of the pool to be constant. Since we did not have access to this file, we can not check if that is done correctly.

Recommendation

Consider using upgradable contracts or have a solid upgrade plan that is well-tested before an emergency situation occurs.

5.13 Mint function in the Infinity pool will emit the incorrect value. Minor ✓ Fixed

Resolution

Fixed by emitting the right value.

Description

In the `InifinityPool` file the `mint` function recomputes the amount of the assets before emitting the event. While this is fine in a lot of cases, that will not always be true. The result of `previewMint` and `convertToAssets` will only be equal while the `totalAssets` and `totalSupply` are equal. For example, this assumption will break after the first liquidation.

Examples

code/src/Pool/InfinityPool.sol:L449-L457

```
function mint(uint256 shares, address receiver) public isOpen returns (uint256 assets) {
    if(shares == 0) revert InvalidParams();
    // These transfers need to happen before the mint, and this is forcing a higher degree of coupling than is ideal
    assets = previewMint(shares);
    asset.transferFrom(msg.sender, address(this), assets);
    liquidStakingToken.mint(receiver, shares);
    assets = convertToAssets(shares);
    emit Deposit(msg.sender, receiver, assets, shares);
}
```

Recommendation

Use the `assets` value computed by the `previewMint` when emitting the event.

5.14 Incorrect Operator Used Minor ✓ Fixed

Resolution

Fixed.

Description

Minor typo in the `InfinityPool` where the `-=` should be replaced with `-`.

Examples

code/src/Pool/InfinityPool.sol:L200

```
return balance -= feesCollected;
```

5.15 Potential overpayment due to rounding imprecision Minor Won't Fix

Resolution

The issue is acknowledged and the potential loss is considered tolerable.

Description

Inside the `InifintyPool` the `pay` function might accept unaccounted files. Imagine a situation where an Agent is trying to repay only the fees portion of the debt. In that case, the following branch will be executed:

code/src/Pool/InfinityPool.sol:L373-L381

```
if (vc.value <= interestOwed) {
    // compute the amount of epochs this payment covers
    // vc.value is not WAD yet, so divWadDown cancels the extra WAD in interestPerEpoch
    uint256 epochsForward = vc.value.divWadDown(interestPerEpoch);
    // update the account's `epochsPaid` cursor
    account.epochsPaid += epochsForward;
    // since the entire payment is interest, the entire payment is used to compute the fee (principal payments are fee-free)
    feeBasis = vc.value;
} else {
```

The issue is if the `value` does not divide by the `interestPerEpoch` exactly, any remainder will remain in the InfinityPool.

code/src/Pool/InfinityPool.sol:L376

```
uint256 epochsForward = vc.value.divWadDown(interestPerEpoch);
```

Recommendation

Since the remainder will most likely not be too large this is not critical, but ideally, those remaining funds would be included in the `refund` variable.

5.16 `jumpStartAccount` should be subject to the same approval checks as regular borrow. Minor ✓ Fixed

Resolution

Will not be fixed due to the complexity of the fix which will require passing verified credentials to be executed.

Description

`InfinityPool` contract has the ability to kick start an account that will have a debt position in this pool.

Examples

code/src/Pool/InfinityPool.sol:L673-L689

```
function jumpStartAccount(address receiver, uint256 agentID, uint256 accountPrincipal) external onlyOwner {
    Account memory account = _getAccount(agentID);
    // if the account is already initialized, revert
    if (account.principal != 0) revert InvalidState();
    // create the account
    account.principal = accountPrincipal;
    account.startEpoch = block.number;
    account.epochsPaid = block.number;
    // save the account
    account.save(router, agentID, id);
    // add the pool to the agent's list of borrowed pools
    GetRoute.agentPolice(router).addPoolToList(agentID, id);
    // mint the iFIL to the receiver, using principal as the deposit amount
    liquidStakingToken.mint(receiver, convertToShares(accountPrincipal));
    // account for the new principal in the total borrowed of the pool
    totalBorrowed += accountPrincipal;
}
```

Recommendation

We suggest that this action is subject to the same rules as the standard borrow action. Thus checks on DTE, LTV and DTI should be done if possible.

5.17 No `Miner` migration is happening in the current implementation of the `Agent` Acknowledged

Description

All miners should be transferred from the old Agent to a new one when upgrading an Agent. To do so, the new Agent is supposed to call the `migrateMiner` function for every miner:

code/src/Agent/Agent.sol:L219-L235

```
function migrateMiner(uint64 miner) external {
  if (newAgent != msg.sender) revert Unauthorized();
  uint256 newId = IAgent(newAgent).id();
  if (
    // first check to make sure the agentFactory knows about this "agent"
    GetRoute.agentFactory(router).agents(newAgent) != newId ||
    // then make sure this is the same agent, just upgraded
    newId != id ||
    // check to ensure this miner was registered to the original agent
    !minerRegistry.minerRegistered(id, miner)
  ) revert Unauthorized();

  // propose an ownership change (must be accepted in v2 agent)
  miner.changeOwnerAddress(newAgent);

  emit MigrateMiner(msg.sender, newAgent, miner);
}
```

The problem is that this function is not called in the current Agent implementation. Since it's just the first version of an Agent contract, it's not a big issue. There is only one edge case where this may be a vulnerability. That may happen if the owner of an Agent decides to upgrade the contract to the same version. It is possible to do, and in that case, the miners' funds will be lost.

Recommendation

It's important to remember to call `migrateMiner` in a new version and not allow upgrading to the same implementation.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
code/src/Router/GetRoute.sol	6daec127e02969538f425c58e6ca9ecef9bc4db5
code/src/Router/Router.sol	0685f3bf844c53bb0e659d97863776cd93232ab8
code/src/Credentials/CredParser.sol	b2a63105b1e9d661191e0c66c9edca7571bbeb88
code/src/Pool/RateModule.sol	9b1440a9a162e83a2842592d6396d27c14c03f5d
code/src/Pool/InfinityPool.sol	5b0774d40a66d267cd61a4c37c66039c7b7d0f5b
code/src/Pool/Account.sol	51dceec029f4cff019324a24af2e601754ac1a4e
code/src/Pool/PoolRegistry.sol	cdddeadd14f0a06789574834866b1f9fee98c3d3
code/src/OffRamp/OffRamp.sol	762ffaf1e056971cd6ec28e20034f9d26ada7d14
code/src/Ping.sol	722e3a96f7bd2c95a5222c6d71a1e3e1805849a2
code/src/Agent/AgentDeployer.sol	5955c1baf1d4a4a4033af6f6aef4d7d9e8aac22a
code/src/Agent/AgentFactory.sol	3f6409e8539aea04a3470892d8e6ebe565c9cfd8
code/src/Agent/AgentPolice.sol	d97389ef65f7454b2f07f55977b97a127ea9ce0b
code/src/Agent/MinerRegistry.sol	2694498172c5bf7437b15639c6a8c117c5721795
code/src/Agent/Agent.sol	60615e447a530d7c75e00ed6555b49a1cc2d8067
code/src/Constants/Epochs.sol	18a0ec65da645d7d05815d53e8aace210b4131a0
code/src/Constants/Routes.sol	8048a5a27d3c15c7c900a460ae7679df1b1c1efe

File	SHA-1 hash
code/src/Auth/Operatable.sol	bc7dd5c1e38d44a00edea5eecd03d6782681dcc9
code/src/VCVerifier/VCVerifier.sol	779b4b371db9eaea011fdfc76c372e3f4b119a39
code/src/Auth/AuthController.sol	16832e4269494bf27e76f3378ed600554a042ea0
code/src/Auth/Ownable.sol	5d19f86e7d61d2ef5372000a39bca6c3c4cfebfe
code/src/Types/Interfaces/IRouter.sol	4b7f9c9c3ba75a492826e4bdd3892c4d706e93ad
code/src/Types/Interfaces/IPoolDeployer.sol	621e6fbc2784b5c62b415b953d9a7cde7b983462
code/src/Types/Interfaces/IERC4626.sol	d0e4c0bd84830bebebfb5edfc80e7c986063a850
code/src/Types/Interfaces/IPoolToken.sol	7947c77739d58e18d3951b91489ee990f20770ab
code/src/Types/Interfaces/IWFIL.sol	5f1126c9c67d9b37911e87c89f1960b55e35e81b
code/src/Types/Interfaces/IPool.sol	a3a134716825a01c6b7a763096cb6c7f0484d763
code/src/Types/Interfaces/IOffRamp.sol	8dcfeef23ec2e18bda2cb493a2977b3c6f63afb2
code/src/Types/Interfaces/IRateModule.sol	fb9b4c66c875c857ea8463df57f1db8b0df4ac55
code/src/Types/Interfaces/IMinerRegistry.sol	fb68f01e13eefb983ce11f563346347ca227fba1
code/src/Types/Interfaces/IAuth.sol	e01dee1cbc4b6f06138cb7b0cbea8863304eb82f
code/src/Types/Interfaces/IAgentFactory.sol	4f1043dad394f463d83f677dfd8bb1f7fc18407d
code/src/Types/Interfaces/IVCVerifier.sol	e0ca19720aba9706ba1b34838d09424c5bd4ead4
code/src/Types/Interfaces/IInfinityPool.sol	eda51ffa1d28a1a74385919e812f78510b574b56
code/src/Types/Interfaces/IPoolTokenPlus.sol	95bf761a0193943f86c4039c1833c096064cd52c
code/src/Types/Interfaces/IAgent.sol	d490fa07cb644d88dbb6df9538a42f0256360e6d
code/src/Types/Interfaces/IPoolRegistry.sol	bc9507b8cac85e1b092e1a98e755942b5f5280ab
code/src/Types/Interfaces/IAgentPolice.sol	7cf78393300ccef96c3bbb74dcd22aa5b3a62314
code/src/Types/Interfaces/IAgentDeployer.sol	0e01ccb2420db5f934870d36900c101f666c9cd1
code/src/Types/Interfaces/IERC20.sol	3f8f9d66083281998547ead9e2a599f5e3d049f8
code/src/Types/Interfaces/ICredentials.sol	bdda5c48fcbe34ba8d4ab64e87bfaf51873d726e
code/src/Types/Structs/Credentials.sol	bdfa1dcec12d187719fe4b02bc4a42e713f2ef6d
code/src/Types/Structs/Account.sol	2bef81b3f52d6f91ef02a8373ea015ca4d33249a
code/src/Types/Structs/Beneficiary.sol	36635042594c7b8b73a80dbabaecd63c7bf423d7

Appendix 2 - Disclosure