SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: GLIF
Prepared by: Sherlock
Lead Security Expert: stopthecap
Dates Audited: June 30 - July 11, 2023
Prepared on: August 23, 2023

Introduction

The Swiss Army Knife of @Filecoin DeFi Developed GLIF Pools - Filecoin's premier staking protocol.

Scope

Repository: glifio/pools

Branch: primary

Commit: e7c43ea72687930aa6fa9822c57f1730b087388f

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium 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.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
17	1

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

deadrxsezzz
rvierdiiev
GimelSec
stopthecap
0xdeadbeef
CodingNameKiki
n33k
carrotsmuggler
VAD37
neumo
ak1
minhtrng

Issue H-1: Funds not withdrawable because `configuredForTakeover` allows to add miners with expired beneficiary

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/66

Found by

0xdeadbeef

Summary

`configuredForTakeover` which is called when a user adds a miner to the agent allows to add miners with expired beneficiary addresses. However - there is no functionality in the agent to change the beneficiary to the agent address. Therefore - an agent will not be able to call `pullFunds` from the miner which could lead to inability to pay pool debt as expected and repay protocol debt therefore lose of funds.

Vulnerability Detail

The agent owner calls `addMiner` to add a new miner https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Agent/Agent.sol#L155

```
function addMiner(
  SignedCredential calldata sc
) external onlyOwnerOperator validateAndBurnCred(sc) checkVersion {
  // confirm the miner is valid and can be added
  if (!sc.vc.target.configuredForTakeover()) revert Unauthorized();
  // change the owner address
  sc.vc.target.changeOwnerAddress(address(this));
  // add the miner to the central registry, this call will revert if the miner
 ↪  is already registered
  minerRegistry.addMiner(id, sc.vc.target);
}
```

https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/shim/FEVM/MinerHelper.sol#L80 `configuredForTakeover` actively permits miners that have an expired beneficiary

```
function configuredForTakeover(uint64 target) internal returns (bool) {
  CommonTypes.FilActorId minerId = _getMinerId(target);
  MinerTypes.GetBeneficiaryReturn memory ret = MinerAPI.getBeneficiary(minerId);

  // if the beneficiary address is the miner's owner, then the agent will assume
 ↪  beneficiary
```

```
  if (_getOwner(minerId) ==
↪   PrecompilesAPI.resolveAddress(ret.active.beneficiary)) {
    return true;
  }

  // if the beneficiary address is expired, then Agent will be ok to take
↪   ownership
  MinerTypes.BeneficiaryTerm memory term = ret.active.term;
  uint256 expiration =
↪   uint256(uint64(CommonTypes.ChainEpoch.unwrap(term.expiration)));
  if (expiration < block.number) return true;

  return false;
}
```

The issue is that in such case the agent owner cannot call `pullFunds` since the miner `withdrawBalance` will fail as the owner is not the beneficiary and the beneficiary has expired: https://github.com/filecoin-project/builtin-actors/blob/master/actors/miner/src/lib.rs#L3293

Impact

Agent will not be able to:

1. withdraw funds from miner to agent. This can lead to inability to pay debt to pool
2. repay debt to the protocol (lead to more slashing)

Code Snippet

Tool used

Manual Review

Recommendation

Either:

1. Not allow expired beneficiary
2. Add a `changeBeneficiary` function in the agent that calls the miner `changeBeneficiary()` function with the agent address

Discussion

Schwartz10

This is a good find.

0xdeadbeef0x

Escalate

This should be HIGH

Reasoning: (1) Expired beneficiary are explicitly supported by the protocol

```
  function configuredForTakeover(uint64 target) internal returns (bool) {
--------
    // if the beneficiary address is expired, then Agent will be ok to take
↪   ownership
    MinerTypes.BeneficiaryTerm memory term = ret.active.term;
    uint256 expiration =
↪   uint256(uint64(CommonTypes.ChainEpoch.unwrap(term.expiration)));
    if (expiration < block.number) return true;
-----
  }
```

(2) There are no protocol related preconditions. It is reasonable that miners have an expired beneficiary. It is a living state of a miner rather then a precondition (3) Since this is supposed to be supported by the protocol, users will not be able to to pay debt with miner rewards - leading to defaults and loss of funds and possible slashing.

This is not an unlikely scenario which results in loss of funds without any protocol based preconditions. I request to reconsider severity

sherlock-admin2

> Escalate
>
> This should be HIGH
>
> Reasoning: (1) Expired beneficiary are explicitly supported by the protocol
>
> ```
>   function configuredForTakeover(uint64 target) internal returns (bool) {
> --------
>     // if the beneficiary address is expired, then Agent will be ok to take
> ↪   ownership
>     MinerTypes.BeneficiaryTerm memory term = ret.active.term;
>     uint256 expiration =
> ↪   uint256(uint64(CommonTypes.ChainEpoch.unwrap(term.expiration)));
>     if (expiration < block.number) return true;
> -----
>   }
> ```

(2) There are no protocol related preconditions. It is reasonable that miners have an expired beneficiary. It is a living state of a miner rather then a precondition (3) Since this is supposed to be supported by the protocol, users will not be able to to pay debt with miner rewards - leading to defaults and loss of funds and possible slashing.

This is not an unlikely scenario which results in loss of funds without any protocol based preconditions. I request to reconsider severity

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

aktech297

I think this issue is low. `configuredForTakeover` is called by the trusted `onlyOwnerOperator`. In this contest,

```
Q: Is the admin/owner of the protocol/contracts TRUSTED or RESTRICTED?
TRUSTED
```

Admin/Owners by default are trusted with the inputs when initializing or calling a function. So this would be low according to Sherlock rules.

0xffff11

Aktech has a point, but I do stand with a medium here.

Also, the issue has been fixed by deleting the logic on expired beneficiaries, which means that if a beneficiary is expired then it will return false instead of true

Schwartz10

Fixed by https://github.com/glif-confidential/pools/pull/525

aktech297

As per Sherlock rules, even if it is fixed by sponser , if that it does not fit under issue criteria , the submission deemed to be low.

0xdeadbeef0x

Thanks for your inputs.

Every function in an agent has the `onlyOwnerOperator` or `onlyOwner` modifier. The issue in not incorrect input/user error. The input is correct and supported.

Allow me to explain a bit: The problem here is that the protocol ACTIVELY accepts a miner that has an expired beneficiary but cannot support it which leads to loss of funds.

Its like a lending protocol that allows you to harvest yield from your locked collateral (which is deemed supported) (sentiment style) to pay debt but after you deposit your collateral you can't use your yield. Which in this case is a bit more extreme because we are talking about miners that can get slashed and pools that you will not be able to pay.

Lets get the judges input on this.

aktech297

Why I think it is Low. Let me refer one of the recently judged issue where its severity downgraded to Low. https://github.com/sherlock-audit/2023-06-arrakis-judging/issues/179#issuecomment-1667788665

In the above case, there is mistake observed in the function call where the whitelisted caller has incorrect flow. Still this function falls under the owner control, as per sherlock rule and contest's FAQ, it is considered as low.

I do see the same in this issue as well. `configuredForTakeover` is called by the `onlyOwnerOperator` Also, there is mentioned about the bots and input validations in the contest FAQ session.

```
 Are there any off-chain mechanisms or off-chain procedures for the protocol
(keeper bots, input validation expectations, etc)? GLIF runs and solely
controls its own unique oracling system that issues verifiable credentials
to Storage Providers who wish to borrow. Read more here
https://docs.google.com/document/d/1nHpdoUqtPuOGBWZu2BsqiaHle8qY4aTTxRr46bMLXR8/edit#hea
```

0xdeadbeef0x

Im sorry. I do not see any relevance to what you wrote. The owner in this case is the agent owner that simply adds a miner (like all operations an agent can do). there is no incorrect flow. no incorrect values.

As can be seen in the issue description - the protocol allows expired beneficiary when taking over

```
// if the beneficiary address is expired, then Agent will be ok to take ownership
MinerTypes.BeneficiaryTerm memory term = ret.active.term;
uint256 expiration =
↪  uint256(uint64(CommonTypes.ChainEpoch.unwrap(term.expiration)));
if (expiration < block.number) return true;
```

The issue here is that the protocol assumes on takeover (transferring ownership from miner owner to agent contract) that it is ok if there is an expired beneficiary since the agent will be the new beneficiary (or could be changes). However, there is no way to change the beneficiary after taking over and agent will not be able to withdraw the miner rewards/funds.

Again.. nothing to do with owner control, bots, off-chain procedures, oracles, etc..

There is enough information here to make a decision. Please lets not make this thread longer unless asked from judge.

aktech297

The block of code that you referring come under the function `configuredForTakeover` which is called by the `onlyOwnerOperator` . The submission talks about the expired miner which is set by the `onlyOwnerOperator` I clearly see that issue is pointing to incorrect value set by the owner. As per sherlock judgin rules and FAQ, this issue is low.

```
Q: Is the admin/owner of the protocol/contracts TRUSTED or RESTRICTED?
TRUSTED
```

Sure, lets hear from sherlock judges.

MLON33

> Aktech has a point, but I do stand with a medium here.
>
> Also, the issue has been fixed by deleting the logic on expired beneficiaries, which means that if a beneficiary is expired then it will return false instead of true

Above confirms Lead Senior Watson signed-off.

hrishibhat

Result: High Unique After reviewing the issue and the comments, this issue is a valid high. This is not an admin error. `configuredForTakeover` is to validate that the miner can be taken over by the agent. `changeOwnerAddress` is supposed to make agent both the owner and beneficiary. However, in case of expired beneficiary, this does not happen https://github.com/filecoin-project/builtin-actors/blob/dd5dd69a78cf8959b4ff4cd71015904d375a3aec/actors/miner/src/lib.rs#L409-L412 As a result as pointed in the issue Agent cannot `pullfunds` hence the impact described in the issue:

> withdraw funds from miner to agent. This can lead to inability to pay debt to pool repay debt to the protocol (lead to more slashing)

This is a clear vulnerability in the code where expired beneficiaries are expected and this results in inability to pull funds to repay debt.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- 0xdeadbeef0x: accepted

aktech297

Sorry to say.. this way of judging is too funny.. in one contest same situation is treated as low and in other contests it's high.. something really concerning..

Issue M-1: Operator can backrun owner to avoid changing operator

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/6

Found by

rvierdiiev

Summary

Operator can backrun owner to avoid changing operator. It's possible because `transferOperator` function can be called by both of them.

Vulnerability Detail

`Agent` contract extends `Operatable`. This is because owner of agent can delegate maintenance of agent to operator. Because of that owner should be able to change operator when he wants. It's 2 step process: set pending operator and then this operator should approve it. For this purpose owner has `transferOperator` function. https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Auth/Operatable.sol#L56-L58

```
function transferOperator(address newOperator) public virtual onlyOwnerOperator {
    pendingOperator = newOperator;
}
```

The problem is that operator also can call this function. As result operator can call it right after owner proposed new operator in order to set it to 0. In such way operator can not allow owner to change it.

Impact

Changing of operator can be blocked by current operator.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

Make `transferOperator` be callable by owner only.

Discussion

0xffff11

Fixed by removing the `onlyOwnerOperator` and adding the `onlyOwner` modifier instead

Schwartz10

Fixed with https://github.com/glif-confidential/pools/pull/532

Issue M-2: AgentPolice._writeOffPools doesn't consider interests

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/10

Found by

rvierdiiev

Summary

AgentPolice._writeOffPools doesn't consider interests. Because of that agent pays less then he should and pool doesn't earn.

Vulnerability Detail

When agent is liquidated, then all funds that were received from it [will be returned to the pools, he used](https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Agent/AgentPolice.sol#L222.

This function will go through all pools and provide them amount of recovered funds according to the amount that agents owns to pool. https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Agent/AgentPolice.sol#L413-L429

```
for (i = 0; i < poolCount; ++i) {
  principal = AccountHelpers.getAccount(router, _agentID, _pools[i]).principal;
  principalAmts[i] = principal;
  totalPrincipal += principal;
}


for (i = 0; i < poolCount; ++i) {
  poolID = _pools[i];
  // compute this pool's share of the total amount
  poolShare = (principalAmts[i] * _totalAmount / totalPrincipal);
  // approve the pool to pull in WFIL
  IPool pool = GetRoute.pool(poolRegistry, poolID);
  wFIL.approve(address(pool), poolShare);
  // write off the pool's assets
  totalOwed = pool.writeOff(_agentID, poolShare);
  excessFunds += poolShare > totalOwed ? poolShare - totalOwed : 0;
}
```

Note, that `principal` inside account doesn't have any info about accrued interests. It's just borrowed amount.

When `writeOff` for pool is called, then pool uses provided values as payment of agent. Pool expects to receive interests as well, but `AgentPolice` doesn't know how much to send.

Because of that next situation is possible: 1.there are 2 pool that agent used. Both of them has debt 1000.$But one of them has all interests repaid and another one has 5$ debt. 2.for any reason agent is liquidated and there are exactly 2005$ to distribute. 3.share of each pool will be 1002.5 only. 4.first pool will use only 1000$ while another one will use 1002.5. 5.As result 2.5$ will be sent to agent's owner and second pool doesn't receive 2.5$.

Impact

Liquidated funds are not used efficiently.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

You can make oracle calculate principal + interests that agent should pay for each pool and write off according to that info.

Discussion

Schwartz10

The Infinity Pool counts interest it should be owed by the Agent in its calculation: https://github.com/sherlock-audit/2023-06-glif/blob/0f3c9afd6ad4af59621efa6e42fa2b7aebaf525f/pools/src/Pool/InfinityPool.sol#L284

rvierdiyev

Escalate for 10 usdc I believe that sponsor didn't understand the point in this report. Such situation is real and can happen when you have not 1 pool. Kindly ask sponsor to review report.

sherlock-admin2

> Escalate for 10 usdc I believe that sponsor didn't understand the point in this report. Such situation is real and can happen when you have not 1 pool. Kindly ask sponsor to review report.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Schwartz10

Yeah the issue is not "writeOffPools doesn't consider interest" because it does - but the issue where - "there was enough funds to pay back all pools, but not all pools got paid back in full" is real

hrishibhat

@rvierdiyev

rvierdiyev

@hrishibhat sponsor has confirmed issue, but the bad naming of report(my bad english) confused him when he looked into it before.

hrishibhat

Result: Medium Unique Considering this issue a valid medium based on the above comments.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- rvierdiyev: accepted

Issue M-3: when rate will be changed for Pool then users will pay for all previous epochs accrding to new rate

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/14

Found by

rvierdiiev

Summary

When rate will be changed for Pool then users will pay for all previous epochs accrding to new rate

Vulnerability Detail

When agent borrows funds from pool, then is should pay interests according to the rate. Currently this rate is minimum(gcred 100), but it can be changed and then `gcred` of account will be changed and rate will be changed.

Once it will be done, that means that user should pay for the previous epochs using this new rate. This is incorrect. Another point, which should be considered is that when agent is liquidated, then penalty rate is used. And again user pays with this penalty rate for all previous epochs, where he was not malicious. This is also incorrect.

Impact

User pays more fees.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

User should pay for previous epochs according to the previous rates. Same can be said about liquidation and penalty rate.

Issue M-4: `PoolRegistry#upgradePool` should call `SimpleRamp#refres`

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/21

Found by

deadrxsezzz

Summary

After a pool is upgraded, `ramp#refreshExtern` should be called in order to connect it to the new version of the pool. Although it is an external function which anyone can call, an unlucky user might attempt to use the ramp after a pool has been upgraded, but the ramp hasn't yet been refreshed and end up losing their funds.

Vulnerability Detail

Problem 1

After a pool is successfully upgraded, all liquid assets are transferred to its new version. This is important as the ramp calculates the share price based of the assets in the pool + `totalBorrowed`. If the pool is upgraded, but the ramp isn't updated a user withdrawing will receive their assets based only on `totalBorrowed`

```
function redeem(
    uint256 shares,
    address receiver,
    address owner,
    uint256
) public ownerIsCaller(owner) returns (uint256 assets) {
    assets = pool.convertToAssets(shares);
    _processExit(owner, receiver, shares, assets);
}
```

```
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.mulDivDown(totalAssets(), supply);
}
```

```
function totalAssets() public view override returns (uint256) {
    return asset.balanceOf(address(this)) + totalBorrowed +
↪   preStake.totalValueLocked() - feesCollected;   // @audit - returns
↪   totalBorrwed
```

```
}
```

In the case where the old pool has a low amount of `totalBorrowed` the innocent user will burn their `iFIL` and receive very little `wFIL`

Problem 2

Another issue which arises from not calling `refreshExtern` is that if there is `FIL`/`wFIL` in the ramp contract and a user calls `recoverFIL` after the pool has been upgraded, but before `refreshExtern` has been called, the `FIL` from the ramp will be sent to the old pool. from where the funds cannot be later retrieved.

Impact

Loss of funds for an innocent user. Forever stuck funds.

Code Snippet

https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/OffRamp/SimpleRamp.sol#L174C5-L178C6 https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Pool/InfinityPool.sol#L653 https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Pool/PoolRegistry.sol#L98C1-L114C1

Tool used

Manual Review

Recommendation

Add the following line of code to `PoolRegistry#upgradePool`

```
oldPool.ramp.refreshExtern();
```

Discussion

0xffff11

Fixed by adding a modifier that checks the address of the new pool against the one in the registry. If they don't match, don't allow to call certain functions like `redeem`

Schwartz10

https://github.com/glif-confidential/pools/pull/540

Issue M-5: Only `newAgent` can call `Agent.migrateMiner`, but `newAgent` may be able to call it

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/28

Found by

0xdeadbeef, CodingNameKiki, GimelSec, neumo

Summary

If the owner of the agent wants to upgrade the Agent instance, the owner would call `AgentFactory.upgradeAgent`. `AgentFactory.upgradeAgent` then calls `oldAgent.decommissionAgent` to transfer funds from old agent to new agent and set `oldAgent.newAgent` to the new deployed agent instance. After that, the new agent is able to call `oldAgent.migrateMiner` to migrate the miners. However, the new agent is not able to call `migrateMiner` in the current implementation.

Vulnerability Detail

An owner of the agent can call `upgradeAgent` to upgrade the agent instance. It calls `oldAgent.decommissionAgent(newAgent)` https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Agent/AgentFactory.sol#L51

```
function upgradeAgent(
  address agent
) external returns (address newAgent) {

  if ((owner != msg.sender && oldAgent.administration() != msg.sender) ||
↪   agentId == 0) revert Unauthorized();
  // deploy a new instance of Agent with the same ID and auth
  newAgent = agDeployer.deploy(
    router,
    agentId,
    owner,
    IAuth(address(oldAgent)).operator(),
    oldAgent.adoRequestKey()
  );
  // Register the new agent and unregister the old agent
  agents[newAgent] = agentId;
  // delete the old agent from the registry
  agents[agent] = 0;
  // transfer funds from old agent to new agent and mark old agent as
↪   decommissioning
  oldAgent.decommissionAgent(newAgent);
```

```
}
```

`newAgent` is set in `decommissionAgent`. https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Agent/Agent.sol#L203

```
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);
}
```

After `AgentFactory.upgradeAgent`, the new agent is able to call `migrateMiner` to migrate miners. https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Agent/Agent.sol#L217

```
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);
}
```

However, it doesn't have a method in `agent.sol` that can `migrateMiner`. It seems like this function should be called by the owner of `newAgent` instead of `newAgent` itself.

Impact

`Agent.migrateMiner` is actually not callable.

Code Snippet

https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Agent/Agent.sol#L217 https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Agent/Agent.sol#L203 https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Agent/AgentFactory.sol#L51

Tool used

Manual Review

Recommendation

Modify the check so that the owner of the `newAgent` can call the function.

```
  function migrateMiner(uint64 miner) external {
-   if (newAgent != msg.sender) revert Unauthorized();
+   if (newAgent != IAuth(newAgent).owner()) 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);
  }
```

Discussion

Schwartz10

Yeah this is valid - the size constraint of the current Agent contract prevented us from adding more logic that wouldn't be used in *this* version of the Agent. An Upgraded version needs a migrateMiner handler to call migrateMiner on the old Agent

Issue M-6: Bypassing the `isOpen` modifier in the pools

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/33

Found by

0xdeadbeef, GimelSec, neumo, stopthecap

Summary

Bypassing the `isOpen` modifier in the pools

Vulnerability Detail

Currently, GLIF has a modifier `isOpen` that "pauses" the interaction with several functions from the pool, some of them are: borrow , pay , deposit and mint .

This modifier can be set to false for several reasons, upgrading and agent, or simply because there is a security issue and you need to pause the system.

This modifier can be actually bypassed by triggering either `receive` or `fallback` in the pool contract, because they call the internal function to deposit instead of the external one that has the modifier.

Impact

System can still be interacted with when it must be paused. This could cause issues when the contract is paused.

Code Snippet

https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Pool/InfinityPool.sol#L447

Tool used

Manual Review

Recommendation

add the modifier also in the internal function

Discussion

0xffff11

Fixed by adding an `isOpen` to the receive() and fallback() methods

Schwartz10

Fixed by https://github.com/glif-confidential/pools/pull/530

Issue M-7: Agents might not be able to prevent being liquidated (defaulted)

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/35

Found by

GimelSec, deadrxsezzz, rvierdiiev, stopthecap

Summary

Agents might not be able to prevent being liquidated (defaulted)

Vulnerability Detail

GLIF uses a modifier `isOpen` to upgrade the pools and to prevent from interacting with specific functions like `borrow` , pay , or `deposit` .

While the modifier is well implemented in most of the codebase, `isOpen` should not be in the `pay` function. That is basically the function were you repay your principal/borrowed amount + interest.

In the case the the contracts are paused, agents will fail to pay back their borrowed amount falling in default states.

Impact

Agents will fall to repay debt and they will be entering default/administration if any pool is paused and it is not being upgraded

Code Snippet

https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Pool/InfinityPool.sol#L354

Tool used

Manual Review

Recommendation

Do not use `isOpen` in pay()

Discussion

Schwartz10

Duplicate https://github.com/sherlock-audit/2023-06-glif-judging/issues/18

hrishibhat

Copying the fix comment from #18

> Fixed by https://github.com/glif-confidential/pools/pull/531

0xffff11

Fixed by removing the isOpen modifier from pay()

MLON33

Copying the Lead Senior Watson sign-off from #18

> Fixed by removing the isOpen modifier from pay()

Issue M-8: Back-run new deployments / front-run `refreshRoutes`

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/36

Found by

0xdeadbeef, VAD37, n33k, stopthecap

Summary

Back-run new deployments / front-run `refreshRoutes`

Vulnerability Detail

GLIF uses a modular system with a registry and routing to get all the contract from their codebase instead of making it upgradeable. In different contracts you have the function:

```
function refreshRoutes() external {
    poolRegistry = GetRoute.poolRegistry(router);
    agentPolice = GetRoute.agentPolice(router);
    agentFactory = GetRoute.agentFactory(router);
    wFIL = GetRoute.wFIL(router);
}
```

that gets the latest deployed and registered address of the system and stores them again as storage variables of the contract this function is in.

This function exists in: Agent, Agent Police, Miner Registry and Infinity Pool.

When GLIF upgrades one of their contracts in this list, refreshRoutes has to be called in the same transaction to not be able to be front-runned. There are several reasons to why they want to upgrade, could be a security issues that has been patched, could be just improvements to the existing contracts.

When upgrading, you are giving the opportunity to MEVs to realize that either something is wrong or there is some kind of opportunity because both contracts.

Not calling `refreshRoutes()` at the same time you upgrade any of the contracts will open the door to this opportunities

Impact

It depends on what is the reason of the upgrade from any contract upgraded. If it has had an exploit, you could still interact with it, or it might be just that you are able to profit from some states that the contract reached before upgrading

Code Snippet

https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Pool/InfinityPool.sol#L759-L764

Tool used

Manual Review

Recommendation

Call `refreshRoutes()` in the same transaction that you upgrade any of the contracts

Issue M-9: User can sandwich pool `writeOff` and profit off of it.

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/44

Found by

deadrxsezzz, n33k

Summary

A user can scan the mempool for pool writeoffs and profit by front-running and back-running it.

Vulnerability Detail

`iFIL`'s value is based on the ratio of `iFIL` minted and `InfinityPool#totalAssets`. Upon a user write-off, `totalBorrowed` is decreased, decreasing the pool's `totalAssets` (and therefore decreasing `iFIL`'s price). A user can be monitoring the mempool and profit off of user write-offs.

Attack scenario

Let's consider there are only 2 equal stakers in `InfinityPool`. Both of them have 500 iFIL and `totalAssets == 2000` (basically iFIL/wFIL ratio is 1:2) A borrower, whose principal was 200 wFIL, has been liquidated and will now be written off. (For simplicity of the example, let's consider no funds were recovered) 1 of the stakers can do the following to make a profit:

1. Scan the mempool for the write-off transaction.
2. When they see the transaction pending, they front-run it, selling all of their stake (leaving only 500 iFIL in circulation and 1000 wFIL in the pool)
3. The write-off transaction executes and it now lowers `totalBorrowed` by the borrower's principal. This also lowers the pool's `totalAssets`, making them `1000 - 200 = 800`
4. The user can now deposit their 1000 wFIL back and they will be minted `((1000 / 800) * 500) = 625` iFIL

In the end, although the 2 users had equal stakes, the one who didn't take any action is left with 500 iFIL (equal to 800 wFIL) The user who did sandwich the transaction is left with 625 wFIL (equal to 1000 wFIL) The user who performed the sandwich attack not only didn't lose any money (unlike the other user), but will also now earn a bigger % of the fees from `InfinityPool`, although the initial investment of the 2 users was the same.

Impact

A user may profit off from innocent users by performing sandwich attacks.

Code Snippet

https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Pool/InfinityPool.sol#L521C1-L525C6 https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Pool/InfinityPool.sol#L249

Tool used

Manual Review

Recommendation

1 solution would be to lock deposits for a certain time. Another solution would be to set a fee on deposits/ withdrawals to make the attack unprofitable.

Discussion

hrishibhat

@iamjakethehuman I think this is a duplicate of #69 and a low for the same reasons as discussed here: https://github.com/sherlock-audit/2023-06-glif-judging/issues/69#issuecomment-1681072544

iamjakethehuman

I don’t agree with low as this could be repeated endlessly and the losses are significant. Also, sponsor has agreed on Medium severity.

hrishibhat

@iamjakethehuman posting my question from the other post here:

> While this seems like a valid impact for only two depositors, in practice wouldn't the deposit interest earned outweigh the loss from a rare bad debt which is distributed among many depositors and also given that there is a capped limit to the borrow?

iamjakethehuman

Doing this sandwich will not result in less interest earned. If sandwich is done properly, user will accrue interest and will avoid the losses from bad debt

Issue M-10: DOS all signatures and verification from agent-police

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/45

Found by

0xdeadbeef, VAD37, carrotsmuggler, deadrxsezzz, minhtrng, neumo, stopthecap

Summary

DOS all signatures and verification from agent-police

Vulnerability Detail

Agents need credentials to operate in the pools. This credentials are checked in the modifier called `validateAndBurnCred(sc)` in the functions from the agent itself.

Overall, this validation ensures that the signature is correctly issued, not expired, not re-used and used by the correct agent Id.

Well, there is a catch. If we go to the modifier:

```
function _validateAndBurnCred(
  SignedCredential calldata signedCredential
) internal {
  agentPolice.isValidCredential(id, msg.sig, signedCredential);
  agentPolice.registerCredentialUseBlock(signedCredential);
}
```

as you can see they are first validating that the signature is fine and all the checks pass and then registering its usage in agent police:

```
function registerCredentialUseBlock(
  SignedCredential memory sc
) external {
  if (IAgent(msg.sender).id() != sc.vc.subject) revert Unauthorized();
  _credentialUseBlock[createSigKey(sc.v, sc.r, sc.s)] = block.number;
}
```

And here we have the critical mistake. In agentPolice, when registering the signature as used, the only check that GLIF makes is `if (IAgent(msg.sender).id() != sc.vc.subject) revert Unauthorized();` , there is no access control or whatsoever.

The problem with this check: `if (IAgent(msg.sender).id() != sc.vc.subject) revert Unauthorized();` that basically means that the agentId has to be the caller and the same one than the `subject` param, it is faulty due to Phantom Agents.

What are phantom agents? Well, there is no check for any specific address or nothing stored on the router. Therefore, you can create a contract with the `IAgent` Interface and create a function with the same selector as `id()`.

This is the first part of the attack.

Second part, front-run the signatures/transactions from the agents to completely leave them in DOS of repaying their debt or borrowing WFIL.

Image Agent with `id() = 2` wants to pay his debt and calls pay() in its own contract. This agent(the owner of the agent or operator) has to pass the full signature parameters with all the parameters:

`SignedCredential calldata sc`

It will contain the exact parameters that we need, that are the v,r,s params of the signature.

When we got this params through the mempool, we front-run the transaction of the agent to `pay()` his debt and in our own malicious IAgent contract, we specify whatever parameters we want in the faulty `SignedCredential memory sc` but adding the v,r,s from the real agent. This check will go through because we are going to pass a faulty credential with the subject as our fakeID

```
 if (IAgent(msg.sender).id() != sc.vc.subject) revert Unauthorized();
```

and we are done, the v,r,s params will be set as used: `_credentialUseBlock[createSigKey(sc.v, sc.r, sc.s)] = block.number;` and the real agents won't be able to repay is debt

Impact

Full DOS of any agents action such as repaying his debt, which will make him fall in a default state

Code Snippet

https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Agent/Agent.sol#L430

Tool used

CRITICAL AUDITING SIGNATURE VIDEO

Recommendation

In the function:

```
function registerCredentialUseBlock(
    SignedCredential memory sc
  ) external {
    if (IAgent(msg.sender).id() != sc.vc.subject) revert Unauthorized();
    _credentialUseBlock[createSigKey(sc.v, sc.r, sc.s)] = block.number;
  }
```

add specific requirements so that the agent is actually a real agent from the system. Maybe fetching the router to check whether they are in a mapping etc.

Discussion

0xffff11

Fixed by adding an `onlyAgent` modifier to the `registerCredentialUseBlock` function.

Schwartz10

Fixed with https://github.com/glif-confidential/pools/pull/528

Issue M-11: `SimpleRamp` will not burn any excess `iFIL` it has, although it is supposed to do so.

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/49

Found by

carrotsmuggler, deadrxsezzz

Summary

If any excess `iFIL` is sent to `SimpleRamp`, it will not be burned, although it is supposed to do so.

Vulnerability Detail

Based on the comments in `_processExit`, we understand that the ramp is supposed to burn any excess `iFIL` sent to the address in order to prevent from accounting issues.

```
// this contract will burn any excess iFIL it has (this shouldn't happen, but in
↪  case it does, we dont have accounting issues)
uint256 balanceOfBefore = iFIL.balanceOf(address(this));
// pull in the iFIL from the iFIL holder, which will decrease the allowance of
↪  this ramp to spend on behalf of the iFIL holder
iFIL.transferFrom(owner, address(this), iFILToBurn);
// burn the exiter's iFIL tokens (and any additional iFIL tokens that somehow
↪  ended up here)
iFIL.burn(
    address(this),
    iFIL.balanceOf(address(this)) - balanceOfBefore
);
```

However, the current implementation burns just the amount of `iFIL` the user has sent, potentially causing accounting issues

Impact

Contract doesn't work as expected. Potential accounting issues.

Code Snippet

https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/OffRamp/SimpleRamp.sol#L127C9-L135C11

Tool used

Manual Review

Recommendation

Change the burn line of code to the following

```
iFIL.burn(address(this), iFIL.balanceOf(address(this)));
```

Discussion

0xffff11

Fixed by adding a setter function to burn any excess of iFil in the contract

Schwartz10

Fixed by https://github.com/glif-confidential/pools/pull/526

Issue M-12: Agent owners can lose funds if agent is updated.

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/61

Found by

0xdeadbeef, deadrxsezzz

Summary

consider the following scenario: Alice is an agent owner. The agent is in debt to the pool so Alice decides to transfer funds to agent and then call the `pay` function. Just before (or in the same block) Alice sends the funds, a new agent version is made and updateAgent is called.

Once an update happens, almost all operations are frozen on the old agent by the `checkVersion` modifier. The transferred funds will be locked in the old agent without the ability to migrate them to the new agent.

Vulnerability Detail

Almost all operations in the agent require that there is no new agent version present. https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Agent/Agent.sol#L535

```
function _checkVersion() internal view {
    if (GetRoute.agentDeployer(router).version() != version) revert
↪   BadAgentState();
  }
```

Once `agentDeployer` is updated to include a new version, no critical operations can happen. The agent can be updated by the owner or by the `administration`. It is reasonable to assume `upgradeAgent` can happen automatically. For example, the protocol could have an automated script to update all agents that are on administration OR the user can have a config enabled in the agent cli to automatically update.

Once the agent is updated. There is no way to send funds to the new agent.

Impact

Any funds send to the agent after update will be permanently locked.

Code Snippet

Tool used

Manual Review

Recommendation

Similar to allowing migration of miners to the new agent, implement a function that allows the owner to migrate funds to the new agent.

Discussion

Schwartz10

When an Agent gets upgraded, the available balance on the Agent is automatically transferred to the new agent https://github.com/sherlock-audit/2023-06-glif/blob/0f3c9afd6ad4af59621efa6e42fa2b7aebaf525f/pools/src/Agent/Agent.sol#L197

0xdeadbeef0x

Escalate

I request to re-read the submission as it is irrelevant to the fact that the available balance on the Agent is automatically transferred to the new agent.

The issue is about transferring funds (in order to pay debt) to the old agent AFTER an update. This is not a user error since the update can happen without the users knowledge/at the same block

It is also reasonable that funds can be streamed to the old agent (after locking collateral at defi protocols). Migrating funds to new agent should be safe and easy

sherlock-admin2

> Escalate
>
> I request to re-read the submission as it is irrelevant to the fact that the available balance on the Agent is automatically transferred to the new agent.
>
> The issue is about transferring funds (in order to pay debt) to the old agent AFTER an update. This is not a user error since the update can happen without the users knowledge/at the same block
>
> It is also reasonable that funds can be streamed to the old agent (after locking collateral at defi protocols). Migrating funds to new agent should be safe and easy

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

iamjakethehuman

In case watson's escalation is accepted, #48 is a dup and should be marked as such

hrishibhat

@Schwartz10

Schwartz10

Yeah - if someone upgrades their agent, and then sends funds to the old agent the funds are stuck

hrishibhat

Result: Medium Has duplicates Given that the upgrades can be done by administration too, there is possible loss here for the agent owner. Considering this issue a valid medium based on escalation and sponsor comment.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- 0xdeadbeef0x: accepted

Issue M-13: Agent version update resets agent state

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/63

Found by

CodingNameKiki, GimelSec, carrotsmuggler, deadrxsezzz, n33k, rvierdiiev

Summary

During the agent version update, funds and some of the storage data are migrated, but the state variables `faultySectorStartEpoch`, `administration` and `defaulted` are not. As a result, the agent state will be reset to a good standing state.

Vulnerability Detail

AgentFactory updates Agent to a new version by calling `upgradeAgent`. It does not migrate the `faultySectorStartEpoch`, `administration` and `defaulted` variables.

```
function upgradeAgent(
  address agent
) external returns (address newAgent) {
  IAgentDeployer agDeployer = GetRoute.agentDeployer(router);
  IAgent oldAgent = IAgent(agent);

  // can only upgrade to a new version of the agent
  if (agDeployer.version() <= oldAgent.version()) revert Unauthorized();

  address owner = IAuth(address(oldAgent)).owner();
  uint256 agentId = agents[agent];
  // only the Agent's owner can upgrade (unless on administration), and only a
↪  registered agent can be upgraded
  if ((owner != msg.sender && oldAgent.administration() != msg.sender) ||
↪  agentId == 0) revert Unauthorized();
  // deploy a new instance of Agent with the same ID and auth
  newAgent = agDeployer.deploy(
    router,
    agentId,
    owner,
    IAuth(address(oldAgent)).operator(),
    oldAgent.adoRequestKey()
  );
  // Register the new agent and unregister the old agent
  agents[newAgent] = agentId;
  // delete the old agent from the registry
  agents[agent] = 0;
```

```
  // transfer funds from old agent to new agent and mark old agent as
↪  decommissioning
  oldAgent.decommissionAgent(newAgent);
}
```

Impact

Agent owner can reset Agent state if AgentDeployer updates its version.

Code Snippet

https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Agent/AgentFactory.sol#L51-L78

Tool used

Manual Review

Recommendation

The `faultySectorStartEpoch`, `administration` and `defaulted` state variables should be migrated during agent updates.

Discussion

Schwartz10

We decided that when we develop the next version of the Agent, we will also address this issue. It's not an easy issue to fix in isolation, before knowing how the next Agents will operate

Issue M-14: Just-in-Time depositor can take interests without actually depositing to the protocol

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/67

Found by

n33k

Summary

The SPs pay the protocol interests via the `pay` function. The depositor can monitor mempool for `pay` call transactions and steal the interests with sandwich attack.

Vulnerability Detail

The attacker monitors the mempool for `pay` transactions and can launch the following sandwich attack.

1. Frontrun with a transaction that deposits a large amount of assets to InfinityPool. This will mints the attacker a large amount of iFIL shares.
2. The `pay` transaction gets executed and the interests are distributed to depositors. The attack will get a large portion of the interests because he holds a large amount of the iFIL shares.
3. Backrun with a withdraw transaction to exit and profit.

Impact

The attacker can steal a large portion of the protocol income.

The severity is set to high because this is an ongoing impact and will render the protocol useless for depositors.

Code Snippet

https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Pool/InfinityPool.sol#L447-L492

Tool used

Manual Review

Recommendation

Add a lock period from deposit to withdraw.

Issue M-15: Faulty interest calculations force agents to overpay interest

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/78

Found by

VAD37, carrotsmuggler, deadrxsezzz

Summary

Agents can be made to overpay due to faulty interest calculations.

Vulnerability Detail

The function `borrow()` is used by agents to borrow FIL tokens and the function `pay()` is used to pay down their debt. Agents are allowed to increase the amount they have borrowed as long as their accounts are healthy. The interest calculated on borrows is tracked by the variable `account.epochsPaid`, which tracks the last block.number for which the interest has been paid. The interest is calculated as `interest = (principal * (block.number - account.epochsPaid) * interestRate)` and the decimals are adjusted.

The problem is that if an agent borrows, then borrows again before paying, the interest is calculated on the new principal, but the `account.epochsPaid` is not updated. This means that the interest is calculated on the new principal from the last paid block. This can be used to force Agents to overpay interest. The main issue is that when a `borrow()` is called, the account's `epochsPaid` is not updated. This is evident by looking at the code snippet for the same function.

https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Pool/InfinityPool.sol#L312-L340

The bug can be reproduced by the following steps:

1. Alice borrows 1 FIL token. the rate of interest is 18% per year. Her `account.epochsPaid` is set to the current block number. Let's assume Alice is required to pay the interest only once a year.
2. After a year, right before paying interest, Alice borrows another 1 FIL token. She then repays her whole debt.
3. Theoretically, according to the specifications of the protocol, Alice is required to pay 18% on the first FIL tokens she borrowed, and no interest or very little interest on the next FIL token since that loan is closed in the very next block. Thus the effective interest rate would be 18% of 1e18 FIL tokens = 1.8e17 FIL tokens.

4. Practically, on the second borrow the account.principal gets bumped to 2e18 FIL tokens. epochsPaid isn't updated. Thus Alice's interest is calculated as 18% of the principal for 1 year, thus 18% of 2e18 FIL tokens = 3.6e17 FIL tokens. This is double the amount of interest Alice should have paid!

Impact

Agents will be forced to pay more interest than necessary due to faulty interest calculations.

Code Snippet

A POC is provided to demonstrate the bug. The POC is a fork of the original test suite. The POC demonstrates the following steps:

1. Scenario #1: Agent borrows an amount, and pays it off entirely after 100 blocks. The interest amount sent to the treasury is recorded. The interest values are then reset in the pool.
2. Scenario #2: Agent borrows the same amount for the same duration. Then the Agent borrows the same amount again and immediately closes the position. The interest amount sent to the treasury is recorded again.
3. According to normal loaning protocols, the interest in both scenarios should be almost the same since the second loan is held for only 1 block. But the POC demonstrates that the interest charged in the second scenario is more than twice that of the first. This is because the interest is calculated on the new principal, but the `account.epochsPaid` is not updated.

```
function testAttackOverpay() public {
    emit log_string("Scenario 1");
    uint256 amount = WAD;
    // Scenario 1
    agentBorrow(
        agent,
        poolID,
        issueGenericBorrowCred(agentID, borrowAmount)
    );
    vm.roll(block.number + 100);
    Account memory account = AccountHelpers.getAccount(
        router,
        address(agent),
        poolID
    );
    emit log_named_uint(
        "Interest blocks",
        block.number - account.epochsPaid
    );
```

```
    emit log_named_uint("Principal", account.principal);
    agentPay(agent, pool, issueGenericPayCred(agentID, 10 * amount));
    uint256 feesCollected = pool.feesCollected();
    pool.harvestFees(feesCollected);
    emit log_named_uint("feesCollected", feesCollected);

    // Scenario 2
    emit log_string("Scenario 2");
    agentBorrow(
        agent,
        poolID,
        issueGenericBorrowCred(agentID, borrowAmount)
    );
    vm.roll(block.number + 100);
    agentBorrow(
        agent,
        poolID,
        issueGenericBorrowCred(agentID, borrowAmount)
    );
    account = AccountHelpers.getAccount(router, address(agent), poolID);
    emit log_named_uint(
        "Interest blocks",
        block.number - account.epochsPaid
    );
    emit log_named_uint("Principal", account.principal);
    agentPay(agent, pool, issueGenericPayCred(agentID, 10 * amount));
    feesCollected = pool.feesCollected();
    pool.harvestFees(feesCollected);
    emit log_named_uint("feesCollected", feesCollected);
}
```

Terminal output:

```
Running 1 test for test/Pool.t.sol:PoolFeeTests
[PASS] testAttackOverpay() (gas: 1181925)
Logs:
  Scenario 1
  Interest blocks: 100
  Principal: 1000000000000000000
  feesCollected: 2113774733638
  Scenario 2
  Interest blocks: 101
  Principal: 2000000000000000000
  feesCollected: 4269406392695
```

Tool used

Foundry

Recommendation

Record the owed interest and then update `account.epochsPaid` to the current block number once `borrow()` is called. This would prevent the interest from being calculated on the new principal from the past paid block.

Discussion

Schwartz10

This is a good find, and we can shorten the `maxEpochsOwedTolerance` to decrease the potential overpay even further, but the max you would over pay on is 1 day of interest, which would be quite small. And you could only do it once https://github.com/sherlock-audit/2023-06-glif/blob/0f3c9afd6ad4af59621efa6e42fa2b7aebaf525f/pools/src/Pool/InfinityPool.sol#L325

Schwartz10

We should not be able to configure the epochs owed setting in the pool to *more* than 24 hours to avoid screwing this up

hrishibhat

Considering the impact low because of the comment and the figures from the POC https://github.com/sherlock-audit/2023-06-glif-judging/issues/78#issuecomment-1635039431

iamjakethehuman

Escalate for 10 USDC This would be an often reoccurring issue for all borrowers. Although, the max interest overpaid would be for a day, this would be a often reoccurring issue, significantly increasing borrower's interest rates. Considering the fees will be times larger than what they are supposed to be, such thing affecting all borrowers should definitely be of at least Medium severity

sherlock-admin2

> Escalate for 10 USDC This would be an often reoccurring issue for all borrowers. Although, the max interest overpaid would be for a day, this would be a often reoccurring issue, significantly increasing borrower's interest rates. Considering the fees will be times larger than what they are supposed to be, such thing affecting all borrowers should definitely be of at least Medium severity

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

hrishibhat

@iamjakethehuman

> but the max you would over pay on is 1 day of interest, which would be quite small. And you could only do it once

This would still seem like a low but would it be possible to share some numbers to have a sense of the extra pay?

iamjakethehuman

@hrishibhat Watson carrotsmuggler has included numbers and test cases in their PoC, but here's some from me: The test case is as follows:

1. User has taken a borrow for X
2. Just under 24 hours later they've taken a borrow for 4X
3. They immediately repay everything, so no new interest is accrued and all of the interest is for the 24 hours: The difference is ~5x in fees. Since it will be a reoccurring issue for all borrowers I believe M is appropriate. Fees with current logic: 301474505327246 Fees with proper logic: 60357686453578

0xffff11

Fixed by adding the check `if (_maxEpochsOwedTolerance > EPOCHS_IN_DAY) revert InvalidParams();` in both the agentPolice and the InifinityPool that enforces epochsOwedTolerance to be < 1 day

0xffff11

@Schwartz10 bringing this one to your attention for severity discussion

Schwartz10

Fixed by https://github.com/glif-confidential/pools/pull/524

carrotsmuggler2

My POC as well as the other warden's comment shows that there is a loss. The question is whether the loss is insignificant or not. Generally losses are called insignificant if they are capped to a certain value, which in this case it is not. It is the interest for 1 day, which for a loan of 1million USD at 18% is 493 USD. Since the losses are a fixed percentage of the principal, and thus scale with the principal amount, I believe this should be a medium. A larger loss would have qualified this as a high. Thus a medium would be appropriate to bridge the difference between rounding errors (low) and large losses (high).

hrishibhat

Result: Medium Has duplicates Although this can be considered to be on the borderline for low-medium because of the impact. Considering this a medium based on the above comments

- There is a flaw in the code where users overpay for a day
- Although happens only once, this still applies for all users

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- iamjakethehuman: accepted

Issue M-16: Agent can prevent liquidation/administration/default by registering to multiple pools

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/112

Found by

0xdeadbeef, deadrxsezzz, rvierdiiev

Summary

There is not cap to the amount of pools an agent can register to. When the platform will support multiple pools, and agent can register to all of them and cause an out of gas error when the agentPolice iterates over them.

Currently, only the InifinityPool is implemented but if as planned more will be supported this vulnerability will exist.

Vulnerability Detail

There is no cap to the amount of pools an agent can borrow from. https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Pool/PoolRegistry.sol#L122

```
function addPoolToList(uint256 agentID, uint256 pool) external onlyPool(pool) {
  _poolIDs[agentID].push(pool);
}
```

The above function gets called by a pool when the agent borrows from it. An agent can borrow a very small amount from all the pools available in the `PoolRegistry`.

This is an issue since the `agentPolice` cannot iterate over a large number of pools as it will consume a lot of gas an potentially more then the allowed block gas limit in filecoin.

The agent needs to iterate over the agent pools when it needs to liquidate/default or put the agent on administration. This is done through the `onlyWhenBehindTargetEpoch` modifier:

```
modifier onlyWhenBehindTargetEpoch(address agent) {
  if (!_epochsPaidBehindTarget(IAgent(agent).id(), defaultWindow)) {
    revert Unauthorized();
  }
  _;
}

function _epochsPaidBehindTarget(
```

```
    uint256 _agentID,
    uint256 _targetEpoch
  ) internal view returns (bool) {
    uint256[] memory pools = poolRegistry.poolIDs(_agentID);

    for (uint256 i = 0; i < pools.length; ++i) { // @audit if large enough - will
  ↪  revert out of gas
      if (AccountHelpers.getAccount(router, _agentID, pools[i]).epochsPaid <
  ↪  block.number - _targetEpoch) {
        return true;
      }
    }

    return false;
  }
```

Impact

the agent can freely borrow and not payup in time without being defaulted/adminstration/liquidated. Leading to loss of pool funds

Code Snippet

Tool used

Manual Review

Recommendation

Set a limit to the amount of pools that can be borrowed against.

Discussion

Schwartz10

Duplicate with https://github.com/sherlock-audit/2023-06-glif-judging/issues/9

carrotsmuggler2

Escalate for 10 USDC

Should be a duplicate of #9, since the fix is already there, but just not implemented. These should be clubbed together.

sherlock-admin2

> Escalate for 10 USDC

> Should be a duplicate of #9, since the fix is already there, but just not implemented. These should be clubbed together.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xffff11

Fixed by adding a `maxPoolsPerAgent` check when getting approval from the AgentPolice

Schwartz10

Fixed with https://github.com/glif-confidential/pools/pull/522

aktech297

I don't think there is any issue in the current implementation. The issue talks about the future updates. As per sherlock rule, any update in future if that is affecting the contract will be out of scope. The updates need separate audit. So sherlock will consider these situation as out of scope. I think, its informational/Low.

Future updates mean, its update on adding more pools.

aktech297

Also, I see there is min limit in amount of FIL can be borrowed. But the submission assumes that there is small amount which is incorrect assumption. From submission. `The above function gets called by a pool when the agent borrows from it. An agent can borrow a very small amount from all the pools available in the PoolRegistry.`

As per implementation, minimum amount should be borrowed is 1 FIL

From codes :

```
function borrow(VerifiableCredential calldata vc) external isOpen
↪  subjectIsAgentCaller(vc) {
    // 1e18 => 1 FIL, can't borrow less than 1 FIL
    ↪  ---------------------------------------------->>> min 1 FIL needed.
    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);
```

If there are 500 Pools, attacker need to borrow 500 FIL. Not sure if the protocol is planned to support 500 Pools. Looking for judges comments on this.

0xdeadbeef0x

Adding my comments:

1. No future update is needed. The contract `agentPolice` where this bug resides iterates over multiple pools (that is why this bug exists)
2. The bug will be fixed by enforcing the already included `maxPoolsPerAgent` (capping the limit to 10 pools per agent)
3. Where is the number 500 coming from? n any case - out of gas can happen with a lower amount of pools and 500 FIL is not an unrealistic price (currently ~2000$).
4. This is classical "out of gas due to unbounded loop" vulnerability which ends with loss of funds. According to sherlock docs, this is valid:

```
Out of Gas: Issues that result in Out of Gas errors either by the malicious user
↪  filling up the arrays or there is a practical call flow that results in OOG
↪  can be considered a valid medium or in cases of blocking all user funds
↪  forever maybe a valid high.
```

Regarding the escalation that this is a duplicate.

I agree that the core issues of this, #9 and #20 are the same (this will lead to out of gas, number of pools not capped). (Looking at the dups of 9, I do not agree that issue #73 has identified the same impact/core of the issues.)

However the impact stated in the issue also adds the impact of preventing liquidation/administration/default which could be considered **HIGH** severity and is not stated in the argued duplications. Therefore I argue that dups of 9 have partially identified the severity of the issue.

Lets wait for the judge to comment.

aktech297

`When the platform will support multiple pools, and agent can register to all of them and cause an out of gas error when the agentPolice iterates over them.`

How many pools will be created ?? 500 ?? Not sure.. The number of pools to make the OOG is unrealistic.

`An agent can borrow a very small amount from all the pools available in the PoolRegistry.` --- incorrect assumption. There needs to minmum 1 FIL to be borrowed.

Even if there are 500 Pools, traversing them will not result in OOG.

`When the platform will support multiple pools` --- depend on platform.. not sure again. probably, the platform can clarify the number of pools they planned.

As I said, even if 500 pools are there, OOG is not possible.

The submission is over exaggerated.

aktech297

Adding few more points

The restriction in number of pools is need not for the OOG prevention, but to limit the number of pools that agent can borrow. This is to make sure that agent will not borrow from all the pools. When agent start to default, all pools will be impacted. This is main reason for limiting the number of pools. As i said, the limitation is not to prevent OOG, since there are no possibility.

0xdeadbeef0x

According to the docs: `pool implementations can be templatized, allowing anyone to deploy a parameterized instance of that particular pool implementation.` - A malicious user can deploy multiple pools and prevent liquidations from any real value pool. The `agentPolice` is generalized to support multiple pools.

I stand by my argument and have commented enough info on this thread. I will wait for the judges decision.

aktech297

can you please show where the pools are deployed by parametrizing and registering by the malicious user themselves.. I don't see any implementation for that.

0xffff11

> According to the docs: `pool implementations can be templatized, allowing anyone to deploy a parameterized instance of that particular pool implementation.` - A malicious user can deploy multiple pools and prevent liquidations from any real value pool. The `agentPolice` is generalized to support multiple pools.
>
> I stand by my argument and have commented enough info on this thread. I will wait for the judges decision.

Could you please show the implementation of the following statement? Thanks

0xdeadbeef0x

Some points to summarize my argument and answer remaining questions:

- I used the provided statement from the GLIF lightpaper to shows GLIF intent for the amount of pools. (noting that Sherlock **does** reward issues that result of future implementations if they are intended).
- Regardless of the statement/future - the current `agentPolice` smart contract is vulnerable. The contract is built to support all pools in the `poolRegistry`. It could be one it could be many. There will be no need to update the

`agentPolice` as the amount of pools increases (and therefore not perform another audit). No future update is required in vulnerable contract.

- The impact of this submission is higher then the severity then #9 and their underlying duplicates even if the core issue is the same. Since the severity is different - the judge could consider not clubbing it with the duplicates for the reward.
- This type of issue is historically rewarded by Sherlock as medium or high.

As I said in my previous comment - there is enough info on this thread to make a decision or request more information. I will answer the judges requests/questions if any.

carrotsmuggler2

This is an exact dupe of #20, which is marked a dupe of #9. This should not be judged separately. #20 even identifies that implementing `maxPoolsPerAgent` will prevent this.

Also the protocol docs clearly mentioned their intention to make pool deployments permissionless in the future, and so it should be a valid concern, even if not implemented yet. `Folks can deploy their own versions customizing the exposed parameters.`

Attackers can deploy a massive number of pools themselves so that the liquidations revert. So attackers can control how many pools get deployed in the whole system.

hrishibhat

Result: Medium Has duplicates Although this issue does describe am impact differently. This is a valid duplicate of #9 and #20 for the same underlying gas reasoning. Keeping this as a main issue for better impact.

I agree that #73 does not identify an issue clearly, although it says `maxPoolsPerAgent` is not enforced, does not identify the problem with not enforcing it.

> In the event of bad debt, this will affect the protocol badly.

the comment seems too vague/informational as against the other issues mentioned above.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- carrotsmuggler2: accepted

aktech297

It's not vague to say this "In the event of bad debt, this will affect the protocol badly." It says when the borrowing is happening in multiple pools, and someone starts defaulting in one way other like not paying or taking loan from too many pools or causing DOS as mentioned in this issue(not confirmed on number of pools ,still the judging is done just by assumption), it's going to affect it badly. When everyone interpret the impact on their own way when there are no clear document, it's really vague to treat the submission as medium just by assumption. Better confirm with sponser about how many number of pools are planned. imo, the judging is done based on documentation.as per Sherlock rules , any judging based on documents is treated as informational or low.

aktech297

When the team is decided is stop with 100 pools, do you think dos is possible? You are judging it just by assumption To me, bad dept is reasonable Dos is still false posting till you don't have any concrete evidence bad debt would like the below one, When the agent borrow from 100 pools , if he can not pay, don't you think it's bad dept which affect the protocol badly I would suggest to check with sponser about it on number of pools case

Issue M-17: Shutting down a pool will prevent exits

Source: https://github.com/sherlock-audit/2023-06-glif-judging/issues/116

Found by

0xdeadbeef, ak1, deadrxsezzz

Summary

The `InfinityPool` has the ability to shutdown. This ability is used to prevent deposits and borrows.

```
/// @dev `isShuttingDown` is a boolean that, when true, halts deposits and
 ↪  borrows. Once set, it cannot be unset.
bool public isShuttingDown = false;
```

It is reasonable to shutdown the pool if it has a fundamental error (that an update will not sole) or to update.

However - when the pool is shutting down also withdrawals/redeem from the ramp would be prevented.

Vulnerability Detail

To withdraw from the `SimpleRamp` the staker calls the withdraw function.

```
function withdraw(
    uint256 assets,
    address receiver,
    address owner,
    uint256
) public ownerIsCaller(owner) returns (uint256 shares) {
    shares = pool.convertToShares(assets);
    _processExit(owner, receiver, shares, assets);
}
```

`_processExit` will then be called

```
    function _processExit(
        address owner,
        address receiver,
        uint256 iFILToBurn,
        uint256 assetsToReceive
    ) internal {
        // if the pool can't process the entire exit, it reverts
```

```
        if (assetsToReceive > pool.getLiquidAssets())
            revert InsufficientLiquidity();
---------
    }
```

As can be seen above the code checks to see if the amount of assets to receive is larger then the pool liquidity of the pool. This is done by calling `pool.getLiquidAssets()`

```
    function getLiquidAssets() public view returns (uint256) {
        if(isShuttingDown) return 0;
--------
    }
```

If the pool is shutting down `getLiquidAssets` will return 0 and the transaction will revert.

Impact

Users will not be able to withdraw their funds. If the pool will not update - their funds will be stuck in the pool.

Code Snippet

Tool used

Manual Review

Recommendation

Consider removing the `if(isShuttingDown) return 0` statement https://github.com/sherlock-audit/2023-06-glif/blob/main/pools/src/Pool/InfinityPool.sol#L208

Discussion

0xffff11

Fixed by deleting `if(isShuttingDown) return 0;` to allow withdrawals

Schwartz10

https://github.com/glif-confidential/pools/pull/521