SHERLOCK

# SHERLOCK SECURITY REVIEW FOR


**Prepared for:** Hats
**Prepared by:** Sherlock
**Lead Security Expert:** obront
**Dates Audited:** February 27 - March 6, 2023
**Prepared on:** April 13, 2023

# Introduction

Hats Protocol is the DAO-native way to structure your organization, empowering contributors with the context, authorities, and accountabilities they need to get things done

## Scope

hats-protocol @ fafcfdf046c0369c1f9e077eacd94a328f9d7af0

- hats-protocol/src/Hats.sol
- hats-protocol/src/HatsIdUtilities.sol

hats-zodiac @ 9455cc0957762f5dbbd8e62063d970199109b977

- hats-zodiac/src/HatsSignerGate.sol
- hats-zodiac/src/HatsSignerGateBase.sol
- hats-zodiac/src/HatsSignerGateFactory.sol
- hats-zodiac/src/MultiHatsSignerGate.sol

## 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 | 9 |

## Issues not fixed or acknowledged

| Medium | High |
|---|---|
| 0 | 0 |

## Security experts who found valid issues

obront
roguereddwarf
cducrest-brainbot
unforgiven
bin2chen
Dug
GimelSec
minhtrng
Allarious
duc
carrot
0xMojito
cccz
james_wu
Bauer
Met
juancito
ktg
rvierdiiev
chaduke
Ace-30

# Issue H-1: Unlinked tophat retains linkedTreeRequests, can be rugged

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/35

## Found by

obront, unforgiven, rogueреddwarf, cducrest-brainbot

## Summary

When a tophat is unlinked from its admin, it is intended to regain its status as a tophat that is fully self-sovereign. However, because the `linkedTreeRequests` value isn't deleted, an independent tophat could still be vulnerable to "takeover" from another admin and could lose its sovereignty.

## Vulnerability Detail

For a tophat to get linked to a new tree, it calls `requestLinkTopHatToTree()` function:

```
function requestLinkTopHatToTree(uint32 _topHatDomain, uint256
↪  _requestedAdminHat) external {
    uint256 fullTopHatId = uint256(_topHatDomain) << 224; // (256 -
↪  TOPHAT_ADDRESS_SPACE);

    _checkAdmin(fullTopHatId);

    linkedTreeRequests[_topHatDomain] = _requestedAdminHat;
    emit TopHatLinkRequested(_topHatDomain, _requestedAdminHat);
}
```

This creates a "request" to link to a given admin, which can later be approved by the admin in question:

```
function approveLinkTopHatToTree(uint32 _topHatDomain, uint256 _newAdminHat)
↪  external {
    // for everything but the last hat level, check the admin of
↪  `_newAdminHat`'s theoretical child hat, since either wearer or admin of
↪  `_newAdminHat` can approve
    if (getHatLevel(_newAdminHat) < MAX_LEVELS) {
        _checkAdmin(buildHatId(_newAdminHat, 1));
    } else {
        // the above buildHatId trick doesn't work for the last hat level, so we
↪  need to explicitly check both admin and wearer in this case
        _checkAdminOrWearer(_newAdminHat);
    }
```

```
    // Linkages must be initiated by a request
    if (_newAdminHat != linkedTreeRequests[_topHatDomain]) revert
↪   LinkageNotRequested();

    // remove the request -- ensures all linkages are initialized by unique
↪   requests,
    // except for relinks (see `relinkTopHatWithinTree`)
    delete linkedTreeRequests[_topHatDomain];

    // execute the link. Replaces existing link, if any.
    _linkTopHatToTree(_topHatDomain, _newAdminHat);
}
```

This function shows that if there is a pending `linkedTreeRequests`, then the admin can use that to link the tophat into their tree and claim authority over it.

When a tophat is unlinked, it is expected to regain its sovereignty:

```
function unlinkTopHatFromTree(uint32 _topHatDomain) external {
    uint256 fullTopHatId = uint256(_topHatDomain) << 224; // (256 -
↪   TOPHAT_ADDRESS_SPACE);
    _checkAdmin(fullTopHatId);

    delete linkedTreeAdmins[_topHatDomain];
    emit TopHatLinked(_topHatDomain, 0);
}
```

However, this function does not delete `linkedTreeRequests`.

Therefore, the following set of actions is possible:

- TopHat is linked to Admin A
- Admin A agrees to unlink the tophat
- Admin A calls `requestLinkTopHatToTree` with any address as the admin
- This call succeeds because Admin A is currently an admin for TopHat
- Admin A unlinks TopHat as promised
- In the future, the address chosen can call `approveLinkTopHatToTree` and take over admin controls for the TopHat without the TopHat's permission

## Impact

Tophats that expect to be fully self-sovereign and without any oversight can be surprisingly claimed by another admin, because settings from a previous admin remain through unlinking.

## Code Snippet

https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/Hats.sol#L688-L732

## Tool used

Manual Review

## Recommendation

In `unlinkTopHatFromTree()`, the `linkedTreeRequests` should be deleted:

```
function unlinkTopHatFromTree(uint32 _topHatDomain) external {
    uint256 fullTopHatId = uint256(_topHatDomain) << 224; // (256 -
↪   TOPHAT_ADDRESS_SPACE);
    _checkAdmin(fullTopHatId);

    delete linkedTreeAdmins[_topHatDomain];
+   delete linkedTreeRequests[_topHatDomain];
    emit TopHatLinked(_topHatDomain, 0);
}
```

## Discussion

**spengrah**

https://github.com/Hats-Protocol/hats-protocol/pull/113

# Issue H-2: Safe can be bricked because threshold is updated with validSignerCount instead of newThreshold

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/37

## Found by

cccz, Dug, duc, Met, obront, roguereddwarf, Allarious, GimelSec, carrot

## Summary

The safe's threshold is supposed to be set with the lower value of the `validSignerCount` and the `targetThreshold` (intended to serve as the maximum). However, the wrong value is used in the call to the safe's function, which in some circumstances can lead to the safe being permanently bricked.

## Vulnerability Detail

In `reconcileSignerCount()`, the valid signer count is calculated. We then create a value called `newThreshold`, and set it to the minimum of the valid signer count and the target threshold. This is intended to be the value that we update the safe's threshold with.

```
if (validSignerCount <= target && validSignerCount != currentThreshold) {
    newThreshold = validSignerCount;
} else if (validSignerCount > target && currentThreshold < target) {
    newThreshold = target;
}
```

However, there is a typo in the contract call, which accidentally uses `validSignerCount` instead of `newThreshold`.

The result is that, if there are more valid signers than the `targetThreshold` that was set, the threshold will be set higher than intended, and the threshold check in `checkAfterExecution()` will fail for being above the max, causing all safe transactions to revert.

This is a major problem because it cannot necessarily be fixed. In the event that it is a gate with a single hat signer, and the eligibility module for the hat doesn't have a way to turn off eligibility, there will be no way to reduce the number of signers. If this number is greater than `maxSigners`, there is no way to increase `targetThreshold` sufficiently to stop the reverting.

The result is that the safe is permanently bricked, and will not be able to perform any transactions.

## Impact

All transactions will revert until `validSignerCount` can be reduced back below `targetThreshold`, which re

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L183-L217

## Tool used

Manual Review

## Recommendation

Change the value in the function call from `validSignerCount` to `newThreshold`.

```
if (newThreshold > 0) {
-    bytes memory data = abi.encodeWithSignature("changeThreshold(uint256)",
↪  validSignerCount);
+    bytes memory data = abi.encodeWithSignature("changeThreshold(uint256)",
↪  newThreshold);

    bool success = safe.execTransactionFromModule(
        address(safe), // to
        0, // value
        data, // data
        Enum.Operation.Call // operation
    );

    if (!success) {
        revert FailedExecChangeThreshold();
    }
}
```

## Discussion

**spengrah**

https://github.com/Hats-Protocol/hats-zodiac/pull/9

# Issue H-3: Signers can bypass checks to add new modules to a safe by abusing reentrancy

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/41

## Found by

obront, roguereddwarf

## Summary

The `checkAfterExecution()` function has checks to ensure that new modules cannot be added by signers. This is a crucial check, because adding a new module could give them unlimited power to make any changes (with no guards in place) in the future. However, by abusing reentrancy, the parameters used by the check can be changed so that this crucial restriction is violated.

## Vulnerability Detail

The `checkAfterExecution()` is intended to uphold important invariants after each signer transaction is completed. This is intended to restrict certain dangerous signer behaviors, the most important of which is adding new modules. This was an issue caught in the previous audit and fixed by comparing the hash of the modules before execution to the has of the modules after.

Before:

```
(address[] memory modules,) = safe.getModulesPaginated(SENTINEL_OWNERS,
↪  enabledModuleCount);
_existingModulesHash = keccak256(abi.encode(modules));
```

After:

```
(address[] memory modules,) = safe.getModulesPaginated(SENTINEL_OWNERS,
↪  enabledModuleCount + 1);
if (keccak256(abi.encode(modules)) != _existingModulesHash) {
    revert SignersCannotChangeModules();
}
```

This is further emphasized in the comments, where it is specified:

> /// @notice Post-flight check to prevent `safe` signers from removing this contract guard, changing any modules, or changing the threshold

## Why Restricting Modules is Important

Modules are the most important thing to check. This is because modules have unlimited power not only to execute transactions but to skip checks in the future. Creating an arbitrary new module is so bad that it is equivalent to the other two issues together: getting complete control over the safe (as if threshold was set to 1) and removing the guard (because they aren't checked in module transactions).

However, this important restriction can be violated by abusing reentrancy into this function.

## Reentrancy Disfunction

To see how this is possible, we first have to take a quick detour regarding reentrancy. It appears that the protocol is attempting to guard against reentrancy with the `guardEntries` variable. It is incremented in `checkTransaction()` (before a transaction is executed) and decremented in `checkAfterExecution()` (after the transaction has completed).

The only protection it provides is in its risk of underflowing, explained in the comments as:

> // leave checked to catch underflows triggered by re-erntry attempts

However, any attempt to reenter and send an additional transaction midstream of another transaction would first trigger the `checkTransaction()` function. This would increment `_guardEntries` and would lead to it not underflowing.

In order for this system to work correctly, the `checkTransaction()` function should simply set `_guardEntries = 1`. This would result in an underflow with the second decrement. But, as it is currently designed, there is no reentrancy protection.

## Using Reentrancy to Bypass Module Check

Remember that the module invariant is upheld by taking a snapshot of the hash of the modules in `checkTransaction()` and saving it in the `_existingModulesHash` variable.

However, imagine the following set of transactions:

- Signers send a transaction via the safe, and modules are snapshotted to `_existingModulesHash`
- The transaction uses the Multicall functionality of the safe, and performs the following actions:
- First, it adds the malicious module to the safe
- Then, it calls `execTransaction()` on itself with any another transaction
- The second call will call `checkTransaction()`

- This will update `_existingModulesHash` to the new list of modules, including the malicious one
- The second call will execute, which doesn't matter (could just be an empty transaction)
- After the transaction, `checkAfterExecution()` will be called, and the modules will match
- After the full transaction is complete, `checkAfterExecution()` will be called for the first transaction, but since `_existingModulesHash` will be overwritten, the module check will pass

## Impact

Any number of signers who are above the threshold will be able to give themselves unlimited access over the safe with no restriction going forward.

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L495-L498

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L522-L525

## Tool used

Manual Review

## Recommendation

Use a more typical reentrancy guard format, such as checking to ensure `_guardEntries == 0` at the top of `checkTransaction()` or simply setting `_guardEntries = 1` in `checkTransaction()` instead of incrementing it.

## Discussion

**zobront**

Escalate for 10 USDC

To successfully duplicate a High Severity issue, it is required for an issue to meet a burden of proof of understanding the exploit.

#67 clearly meets this burden of proof. It explains the same exploit described in this report and deserves to be duplicated with it.

#105 and #124 do not explain any exploit. They simply noticed that the reentrancy guard wouldn't work, couldn't find a way to take advantage of that, and submitted it without a way to use it.

My recommendation is that they are not valid issues, but at the very least they should be moved to a separate Medium issue to account for the fact that they did not find a High Severity exploit.

**sherlock-admin**

> Escalate for 10 USDC
>
> To successfully duplicate a High Severity issue, it is required for an issue to meet a burden of proof of understanding the exploit.
>
> #67 clearly meets this burden of proof. It explains the same exploit described in this report and deserves to be duplicated with it.
>
> #105 and #124 do not explain any exploit. They simply noticed that the reentrancy guard wouldn't work, couldn't find a way to take advantage of that, and submitted it without a way to use it.
>
> My recommendation is that they are not valid issues, but at the very least they should be moved to a separate Medium issue to account for the fact that they did not find a High Severity exploit.

You've created a valid escalation for 10 USDC!

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.

**cducrest**

It's a bit ambitious to have 4 issues describing the same line of codes as incorrect / vulnerable not being marked as duplicate, especially when they provide the same recommendation. I feel like going into such depths to describe the impact may not be necessary to ensure the safety of the protocol.

However, I agree that it can also feel weird that we would be awarded the same while your issue provides much more details. I could not find anything in the Sherlock docs pertaining to this situation, but maybe there should be a reward for the best issue describing a vulnerability.

When first submitting these issues, I feel like I may take the risk that the issue is treated as medium / low by not providing enough details. Perhaps are you already awarded for having provided such details by ensuring your issue is considered valid?

**hrishibhat**

Escalation accepted

Given that issues #41 & #67 have identified a valid attack path, considering #105 & #124 as a medium as it identifies underlying re-entrancy issue.

Note: Sherlock will make note of the above comments and discuss internally to add additional instructions in the guide to help resolve such scenarios in the future.

**sherlock-admin**

> Escalation accepted
>
> Given that issues #41 & #67 have identified a valid attack path, considering #105 & #124 as a medium as it identifies underlying re-entrancy issue.
>
> Note: Sherlock will make note of the above comments and discuss internally to add additional instructions in the guide to help resolve such scenarios in the future.

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

# Issue H-4: If another module adds a module, the safe will be bricked

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/42

## Found by

obront, rogueredwarf, minhtrng

## Summary

If a module is added by another module, it will bypass the `enableNewModule()` function that increments `enabledModuleCount`. This will throw off the module validation in `checkTransaction()` and `checkAfterExecution()` and could cause the safe to become permanently bricked.

## Vulnerability Detail

In order to ensure that signers cannot add new modules to the safe (thus giving them unlimited future governing power), the guard portion of the gate checks that the hash of the modules before the transaction is the same as the hash after.

Before:

```
(address[] memory modules,) = safe.getModulesPaginated(SENTINEL_OWNERS,
↪  enabledModuleCount);
_existingModulesHash = keccak256(abi.encode(modules));
```

After:

```
(address[] memory modules,) = safe.getModulesPaginated(SENTINEL_OWNERS,
↪  enabledModuleCount + 1);
if (keccak256(abi.encode(modules)) != _existingModulesHash) {
    revert SignersCannotChangeModules();
}
```

You'll note that the "before" check uses `enabledModuleCount` and the "after" check uses `enabledModuleCount + 1`. The reason for this is that we want to be able to catch whether the user added a new module, which requires us taking a larger pagination to make sure we can view the additional module.

However, if we were to start with a number of modules larger than `enabledModuleCount`, the result would be that the "before" check would clip off the final modules, and the "after" check would include them, thus leading to different hashes.

This situation can only arise if a module is added that bypasses the `enableModule()` function. But this exact situation can happen if one of the other modules on the safe adds a module to the safe.

In this case, the modules on the safe will increase but `enabledModuleCount` will not. This will lead to the "before" and "after" checks returning different arrays each time, and therefore disallowing transactions.

The only possible ways to fix this problem will be to have the other module remove the additional one they added. But, depending on the specific circumstances, this option may not be possible. For example, the module that performed the adding may not have the ability to remove modules.

## Impact

The safe can be permanently bricked, with the guard functions disallowing any transactions. All funds in the safe will remain permanently stuck.

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L495-L498

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L521-L525

## Tool used

Manual Review

## Recommendation

The module guarding logic needs to be rethought. Given the large number of unbounded risks it opens up, I would recommend not allowing other modules on any safes that use this functionality.

## Discussion

**spengrah**

This is definitely an issue. I can see a few potential approaches to addressing it:

1. As suggested, allow no other modules to exist alongside HSG. That would require checks in the factory as well as in `checkAfterExecution`. This approach would eliminate this issue fully, but would reduce flexibility for DAOs to customize how they use their Safes. For example, a common expected pattern is to use a Safe alongside a DAO that both the DAO (via a module;

slow) and signers (via sigs; fast) can control. This wouldn't be possible if no additional modules could be added.

2. Add an onlyOwner `removeModules` function to enable the owner to true up the module count saved in storage if a module is added to the Safe via a different avenue.
3. In `checkTransaction`, count and store the number of modules that exist on the safe, then use that value as the page size for the check in `checkAfterExecution`. This would allow the Safe to have arbitrary number of modules, but for larger number of modules the gas size be fairly high — in `checkTransaction` we'd need to get the module addresses from the Safe in a while loop to ensure we count all of them.

cc @zobront

**spengrah**

Going with option 1 here

**spengrah**

https://github.com/Hats-Protocol/hats-zodiac/pull/10

# Issue H-5: Other module can add owners to safe that push us above maxSigners, bricking safe

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/46

## Found by

obront, rogueereddwarf

## Summary

If another module adds owners to the safe, these additions are not checked by our module or guard's logic. This can result in pushing us over `maxSigners`, which will cause all transactions to revert. In the case of an immutable hat, the only way to avoid the safe being locked permanently (with all funds frozen) may be to convince many hat wearers to renounce their hats.

## Vulnerability Detail

When new owners are added to the contract through the `claimSigner()` function, the total number of owners is compared to `maxSigners` to ensure it doesn't exceed it.

However, if there are other modules on the safe, they are able to add additional owners without these checks.

In the case of `HatsSignerGate.sol`, there is no need to call `claimSigner()` to "activate" these owners. They will automatically be valid as long as they are a wearer of the correct hat.

This could lead to an issue where many (more than `maxSigners`) wearers of an immutable hat are added to the safe as owners. Now, each time a transaction is processed, `checkTransaction()` is called, which calls `reconcileSignerCount()`, which has the following check:

```
if (validSignerCount > maxSigners) {
    revert MaxSignersReached();
}
```

This will revert.

Worse, there is nothing the admin can do about it. If they don't have control over the eligibility address for the hat, they are not able to burn the hats or transfer them.

The safe will be permanently bricked and unable to perform transactions unless the hat wearers agree to renounce their hats.

## Impact

The safe can be permanently bricked and unable to perform transactions unless the hat wearers agree to renounce their hats.

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L187-L189

## Tool used

Manual Review

## Recommendation

If `validSignerCount > maxSigners`, there should be some mechanism to reduce the number of signers rather than reverting.

Alternatively, as suggested in another issue, to get rid of all the potential risks of having other modules able to make changes outside of your module's logic, we should create the limit that the HatsSignerGate module can only exist on a safe with no other modules.

## Discussion

**spengrah**

I can see a few approaches here

1. Add an onlyOwner function to change `maxSigners`. This opens up more surface area, but would enable the owner to unbrick the safe in this case.
2. Add (and document) a trust assumption that other modules either a) will not add new safe owners, or b) can remove them if they accidentally brick the safe
3. block any modules aside from HSG

cc @zobront

**zobront**

@spengrah I believe that allowing other modules aside from HSG adds a huge risk surface and is better not to. I know there are trade offs, but even if you manage to get everything right, you know there will be mistakes that lead to exploits, and in my view the best thing you can do for users is not allow it.

**spengrah**

https://github.com/Hats-Protocol/hats-zodiac/pull/10

**zobront**

Escalate for 10 USDC

All three dups of this issue (#51, #104, #130) describe the same issue, in which more than `maxSigners` can be added by (a) removing a hat, (b) reconciling, and (c) adding the hat back. This is a valid attack path.

This issue describes a separate issue, in which the extra signer can be added by an external module, which is a totally different attack with a different solution (note: @spengrah please make sure to review the other issues to ensure you have a fix that accounts for them).

This issue should be deduplicated from the other 3, since the attack is totally unrelated and simply results in the same outcome.

**sherlock-admin**

> Escalate for 10 USDC
>
> All three dups of this issue (#51, #104, #130) describe the same issue, in which more than `maxSigners` can be added by (a) removing a hat, (b) reconciling, and (c) adding the hat back. This is a valid attack path.
>
> This issue describes a separate issue, in which the extra signer can be added by an external module, which is a totally different attack with a different solution (note: @spengrah please make sure to review the other issues to ensure you have a fix that accounts for them).
>
> This issue should be deduplicated from the other 3, since the attack is totally unrelated and simply results in the same outcome.

You've created a valid escalation for 10 USDC!

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**

Escalation accepted

Given although the outcome is similar the underlying issues are different Considering #51, #104, and #130 as a separate issue.

**sherlock-admin**

> Escalation accepted
>
> Given although the outcome is similar the underlying issues are different Considering #51, #104, and #130 as a separate issue.

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.



# Issue H-6: Signers can brick safe by adding unlimited additional signers while avoiding checks

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/48

## Found by

obront, rogueredwarf, Dug

## Summary

There are a number of checks in `checkAfterExecution()` to ensure that the signers cannot perform any illegal actions to exert too much control over the safe. However, there is no check to ensure that additional owners are not added to the safe. This could be done in a way that pushes the total over `maxSigners`, which will cause all future transactions to revert.

This means that signers can easily collude to freeze the contract, giving themselves the power to hold the protocol ransom to unfreeze the safe and all funds inside it.

## Vulnerability Detail

When new owners are added to the contract through the `claimSigner()` function, the total number of owners is compared to `maxSigners` to ensure it doesn't exceed it.

However, owners can also be added by a normal `execTransaction` function. In this case, there are very few checks (all of which could easily or accidentally be missed) to stop us from adding too many owners:

```
if (safe.getThreshold() != _getCorrectThreshold()) {
    revert SignersCannotChangeThreshold();
}

function _getCorrectThreshold() internal view returns (uint256 _threshold) {
    uint256 count = _countValidSigners(safe.getOwners());
    uint256 min = minThreshold;
    uint256 max = targetThreshold;
    if (count < min) _threshold = min;
    else if (count > max) _threshold = max;
    else _threshold = count;
}
```

That means that either in the case that (a) the safe's threshold is already at `targetThreshold` or (b) the owners being added are currently toggled off or have eligibility turned off, this check will pass and the owners will be added.

Once they are added, all future transactions will fail. Each time a transaction is processed, `checkTransaction()` is called, which calls `reconcileSignerCount()`, which has the following check:

```
if (validSignerCount > maxSigners) {
    revert MaxSignersReached();
}
```

This will revert as long as the new owners are now activated as valid signers.

In the worst case scenario, valid signers wearing an immutable hat are added as owners when the safe's threshold is already above `targetThreshold`. The check passes, but the new owners are already valid signers. There is no admin action that can revoke the validity of their hats, so the `reconcileSignerCount()` function will always revert, and therefore the safe is unusable.

Since `maxSigners` is immutable and can't be changed, the only solution is for the hat wearers to renounce their hats. Otherwise, the safe will remain unusable with all funds trapped inside.

## Impact

Signers can easily collude to freeze the contract, giving themselves the power to hold the protocol ransom to unfreeze the safe and all funds inside it.

In a less malicious case, signers might accidentally add too many owners and end up needing to manage the logistics of having users renounce their hats.

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L507-L529

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L187-L189

## Tool used

Manual Review

## Recommendation

There should be a check in `checkAfterExecution()` that ensures that the number of owners on the safe has not changed throughout the execution.

It also may be recommended that the `maxSigners` value is adjustable by the contract owner.

## Discussion

**spengrah**

> There is no admin action that can revoke the validity of their hats, so the reconcileSignerCount() function will always revert, and therefore the safe is unusable.

This is not fully true, since either the hat's eligibility or toggle module could revoke their hat. But this is definitely not guaranteed to be possible, especially in the case of mechanistic modules that have hardcoded revocation logic, so addressing this issue is warranted.

My current thinking is that storing/comparing the full owner array — which is likely necessary to address #118 and #70 — would also address this issue, since ensuring exact same owners also ensures same number of owners.

cc @zobront

**spengrah**

https://github.com/Hats-Protocol/hats-zodiac/pull/5

# Issue H-7: HatsSignerGateBase: valid signer threshold can be bypassed because HSG checks signatures differently from Safe which allows exploitation

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/50

## Found by

roguereddwarf, bin2chen

## Summary

This report deals with how the `HatsSignerGate` and the `Safe` check signatures differently which opens the door to exploitation.

I will show how this allows a valid signer that has become invalid but not yet removed from the `owners` of the Safe to continue signing transactions. The invalid signer can effectively sign transactions as though he was valid.

Also there is the possibility of valid signers calling `Safe.addOwnerWithThreshold`. When an owner is added to the Safe but not a valid signer he can still sign transactions and the HSG will not recognize that there are not enough valid signatures.

To summarize, the issue is caused by this:

1. Signatures are first checked by the Safe then by the HSG logic
2. We can pass an arbitrary amount of signatures when executing a transaction
3. The Safe checks that the first `threshold` signatures are valid. However the HSG logic checks that ANY of the signatures are signed by valid signers. The HSG logic does not check the same signatures as the Safe.

Essentially the Safe and HSG logic are applying different checks to different signatures.

## Vulnerability Detail

A transaction is executed by calling Safe.execTransaction

First the signatures are checked by the Safe Link then the `checkTransaction` function is executed on the guard (HatsSignerGate) Link

The HatsSignerGate then executes `countValidSignatures` to check if enough signatures were signed by valid signers Link

With all prerequisites out of the way, we can now get into the actual issue.

The Safe calls `checkNSignatures` to check if the first `threshold` signatures in the `signatures` bytes are valid Link

So if `threshold=5` but we provide say 7 signatures, the last two signatures are not checked. If the first 5 signatures are valid the check passes successfully.

The issue is that the HSG `countValidSignatures` function iterates over ALL signatures and tries to find enough valid signers such that `threshold` is reached Link.

So imagine the following scenario:

1. There are 4 owners in the Safe, `threshold=3` and 3 owners are valid signers.
2. One of the owners is no longer a valid signer (I'll call him Bob). He is not yet removed from the `owners`.
3. Bob wants to sign a transaction and submit it to the Safe. He already has 2 signatures from valid signers.
4. Bob signs the transaction and appends his signature to the `signatures` bytes. He also appends a signature of a valid signer from a previous transaction. So there are now 4 signatures in the `signatures` bytes.
5. Bob calls `Safe.execTransaction`. The Safe checks the first 3 signatures to be valid signatures from owners. The check passes. The HSG checks that there are at least 3 signatures signed by valid signers. Which also passes.

Important: Bob can pass a 4th signature from a previous transaction because two of the signature types used in HSG do not check that the correct data has been signed https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L561-L568.

To summarize: Bob was able to sign a transaction even though he was no longer a valid signer.

## Further notes

Another thing to note is that HSG does not check signatures for uniqueness so if Bob would have to append multiple signatures from valid signers he could just add the same signature multiple times.

Also the HSG does not check that `ecrecover` does not return the zero address as owner which it does if the signature is invalid. These checks are implemented in the Safe. So by implementing the mitigation I suggest below the Safe and HSG will check the same signatures. So there is no need to have these checks in the HSG as well. However due to this bug (checking different signatures), the signer hat might be transferred to `address(0)` which then causes invalid signatures to be considered valid.

## Impact

Owners of the Safe that are not valid signers can sign transactions.

## Code Snippet

Also have a look at the `@audit-info` comments that further explain the issue.

https://github.com/safe-global/safe-contracts/blob/cb22537c89ea4187f4ad141ab2e1abf15b27416b/contracts/Safe.sol#L135-L217

```
function execTransaction(
    address to,
    uint256 value,
    bytes calldata data,
    Enum.Operation operation,
    uint256 safeTxGas,
    uint256 baseGas,
    uint256 gasPrice,
    address gasToken,
    address payable refundReceiver,
    bytes memory signatures
) public payable virtual returns (bool success) {
    bytes32 txHash;
    // Use scope here to limit variable lifetime and prevent `stack too deep`
    ↪ errors
    {
        bytes memory txHashData = encodeTransactionData(
            // Transaction info
            to,
            value,
            data,
            operation,
            safeTxGas,
            // Payment info
            baseGas,
            gasPrice,
            gasToken,
            refundReceiver,
            // Signature info
            nonce
        );
        // Increase nonce and execute transaction.
        nonce++;
        txHash = keccak256(txHashData);
        // @audit-info first the Safe checks the signatures
        checkSignatures(txHash, txHashData, signatures);
    }
```

```
    address guard = getGuard();
    {
        if (guard != address(0)) {
            // @audit-info then signatures are checked by HSG
            Guard(guard).checkTransaction(
                // Transaction info
                to,
                value,
                data,
                operation,
                safeTxGas,
                // Payment info
                baseGas,
                gasPrice,
                gasToken,
                refundReceiver,
                // Signature info
                signatures,
                msg.sender
            );
        }
    }
    // We require some gas to emit the events (at least 2500) after the
↪   execution and some to perform code until the execution (500)
    // We also include the 1/64 in the check that is not send along with a call
↪   to counteract potential shortings because of EIP-150
    require(gasleft() >= ((safeTxGas * 64) / 63).max(safeTxGas + 2500) + 500,
↪   "GS010");
    // Use scope here to limit variable lifetime and prevent `stack too deep`
↪   errors
    {
        uint256 gasUsed = gasleft();
        // If the gasPrice is 0 we assume that nearly all available gas can be
↪   used (it is always more than safeTxGas)
        // We only substract 2500 (compared to the 3000 before) to ensure that
↪   the amount passed is still higher than safeTxGas
        success = execute(to, value, data, operation, gasPrice == 0 ? (gasleft()
↪   - 2500) : safeTxGas);
        gasUsed = gasUsed.sub(gasleft());
        // If no safeTxGas and no gasPrice was set (e.g. both are 0), then the
↪   internal tx is required to be successful
        // This makes it possible to use `estimateGas` without issues, as it
↪   searches for the minimum gas where the tx doesn't revert
        require(success || safeTxGas != 0 || gasPrice != 0, "GS013");
        // We transfer the calculated tx costs to the tx.origin to avoid sending
↪   it to intermediate contracts that have made calls
        uint256 payment = 0;
        if (gasPrice > 0) {
```



```
            payment = handlePayment(gasUsed, baseGas, gasPrice, gasToken,
↪   refundReceiver);
        }
        if (success) emit ExecutionSuccess(txHash, payment);
        else emit ExecutionFailure(txHash, payment);
    }
    {
        if (guard != address(0)) {
            Guard(guard).checkAfterExecution(txHash, success);
        }
    }
}
```

https://github.com/safe-global/safe-contracts/blob/cb22537c89ea4187f4ad141ab2e1abf15b27416b/contracts/Safe.sol#L270-L330

```
// @audit-info requiredSignatures is equal to threshold
function checkNSignatures(bytes32 dataHash, bytes memory data, bytes memory
↪   signatures, uint256 requiredSignatures) public view {
    // Check that the provided signature data is not too short
    require(signatures.length >= requiredSignatures.mul(65), "GS020");
    // There cannot be an owner with address 0.
    address lastOwner = address(0);
    address currentOwner;
    uint8 v;
    bytes32 r;
    bytes32 s;
    uint256 i;
    // @audit-info only the first threshold signatures are checked
    for (i = 0; i < requiredSignatures; i++) {
        (v, r, s) = signatureSplit(signatures, i);
        if (v == 0) {
            require(keccak256(data) == dataHash, "GS027");
            // If v is 0 then it is a contract signature
            // When handling contract signatures the address of the contract is
↪   encoded into r
            currentOwner = address(uint160(uint256(r)));


            // Check that signature data pointer (s) is not pointing inside the
↪   static part of the signatures bytes
            // This check is not completely accurate, since it is possible that
↪   more signatures than the threshold are send.
            // Here we only check that the pointer is not pointing inside the
↪   part that is being processed
            require(uint256(s) >= requiredSignatures.mul(65), "GS021");
```

```
            // Check that signature data pointer (s) is in bounds (points to the
↪  length of data -> 32 bytes)
            require(uint256(s).add(32) <= signatures.length, "GS022");


            // Check if the contract signature is in bounds: start of data is s
↪  + 32 and end is start + signature length
            uint256 contractSignatureLen;
            // solhint-disable-next-line no-inline-assembly
            assembly {
                contractSignatureLen := mload(add(add(signatures, s), 0x20))
            }
            require(uint256(s).add(32).add(contractSignatureLen) <=
↪  signatures.length, "GS023");


            // Check signature
            bytes memory contractSignature;
            // solhint-disable-next-line no-inline-assembly
            assembly {
                // The signature data for contract signatures is appended to the
↪  concatenated signatures and the offset is stored in s
                contractSignature := add(add(signatures, s), 0x20)
            }
            require(ISignatureValidator(currentOwner).isValidSignature(data,
↪  contractSignature) == EIP1271_MAGIC_VALUE, "GS024");
        } else if (v == 1) {
            // If v is 1 then it is an approved hash
            // When handling approved hashes the address of the approver is
↪  encoded into r
            currentOwner = address(uint160(uint256(r)));
            // Hashes are automatically approved by the sender of the message or
↪  when they have been pre-approved via a separate transaction
            require(msg.sender == currentOwner ||
↪  approvedHashes[currentOwner][dataHash] != 0, "GS025");
        } else if (v > 30) {
            // If v > 30 then default va (27,28) has been adjusted for eth_sign
↪  flow
            // To support eth_sign and similar we adjust v and hash the
↪  messageHash with the Ethereum message prefix before applying ecrecover
            currentOwner = ecrecover(keccak256(abi.encodePacked("\x19Ethereum
↪  Signed Message:\n32", dataHash)), v - 4, r, s);
        } else {
            // Default is the ecrecover flow with the provided data hash
            // Use ecrecover with the messageHash for EOA signatures
            currentOwner = ecrecover(dataHash, v, r, s);
        }
```



```
        require(currentOwner > lastOwner && owners[currentOwner] != address(0)
↪   && currentOwner != SENTINEL_OWNERS, "GS026");
        lastOwner = currentOwner;
    }
}
```

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L445-L503

```
function checkTransaction(
    address to,
    uint256 value,
    bytes calldata data,
    Enum.Operation operation,
    uint256 safeTxGas,
    uint256 baseGas,
    uint256 gasPrice,
    address gasToken,
    address payable refundReceiver,
    bytes memory signatures,
    address // msgSender
) external override {
    if (msg.sender != address(safe)) revert NotCalledFromSafe();


    uint256 safeOwnerCount = safe.getOwners().length;
    // uint256 validSignerCount = _countValidSigners(safe.getOwners());


    // ensure that safe threshold is correct
    reconcileSignerCount();


    if (safeOwnerCount < minThreshold) {
        revert BelowMinThreshold(minThreshold, safeOwnerCount);
    }


    // get the tx hash; view function
    bytes32 txHash = safe.getTransactionHash(
        // Transaction info
        to,
        value,
        data,
        operation,
        safeTxGas,
        // Payment info
```

```
        baseGas,
        gasPrice,
        gasToken,
        refundReceiver,
        // Signature info
        // We subtract 1 since nonce was just incremented in the parent function
↪   call
        safe.nonce() - 1 // view function
    );

    // @audit-info all signatures are checked (signatures.length / 65) as
↪   opposed to first threshold ones in the Safe
    uint256 validSigCount = countValidSignatures(txHash, signatures,
↪   signatures.length / 65);


    // revert if there aren't enough valid signatures
    if (validSigCount < safe.getThreshold() || validSigCount < minThreshold) {
        revert InvalidSigners();
    }


    // record existing modules for post-flight check
    // SENTINEL_OWNERS and SENTINEL_MODULES are both address(0x1)
    (address[] memory modules,) = safe.getModulesPaginated(SENTINEL_OWNERS,
↪   enabledModuleCount);
    _existingModulesHash = keccak256(abi.encode(modules));


    unchecked {
        ++_guardEntries;
    }
}
```

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L547-L591

```
function countValidSignatures(bytes32 dataHash, bytes memory signatures, uint256
↪   sigCount)
    public
    view
    returns (uint256 validSigCount)
{
    // There cannot be an owner with address 0.
    address currentOwner;
    uint8 v;
    bytes32 r;
```

```
        bytes32 s;
        uint256 i;

        // @audit-info all signatures are checked
        for (i; i < sigCount;) {
            (v, r, s) = signatureSplit(signatures, i);
            // @audit-info old signature is counted as valid because transaction
↪   data is not verified
            if (v == 0) {
                // If v is 0 then it is a contract signature
                // When handling contract signatures the address of the contract is
↪   encoded into r
                currentOwner = address(uint160(uint256(r)));
            // @audit-info old signature is counted as valid because transaction
↪   data is not verified
            } else if (v == 1) {
                // If v is 1 then it is an approved hash
                // When handling approved hashes the address of the approver is
↪   encoded into r
                currentOwner = address(uint160(uint256(r)));
            } else if (v > 30) {
                // If v > 30 then default va (27,28) has been adjusted for eth_sign
↪   flow
                // To support eth_sign and similar we adjust v and hash the
↪   messageHash with the Ethereum message prefix before applying ecrecover
                currentOwner =
                    ecrecover(keccak256(abi.encodePacked("\x19Ethereum Signed
↪   Message:\n32", dataHash)), v - 4, r, s);
            } else {
                // Default is the ecrecover flow with the provided data hash
                // Use ecrecover with the messageHash for EOA signatures
                currentOwner = ecrecover(dataHash, v, r, s);
            }


            if (isValidSigner(currentOwner)) {
                // shouldn't overflow given reasonable sigCount
                unchecked {
                    ++validSigCount;
                }
            }
            // shouldn't overflow given reasonable sigCount
            unchecked {
                ++i;
            }
        }
    }
```

## Tool used

Manual Review

## Recommendation

I propose that in the HatsSignerGate only the first `threshold` signatures are checked. Such that both the Safe and HSG check the SAME signatures.

Fix:

```
diff --git a/src/HatsSignerGateBase.sol b/src/HatsSignerGateBase.sol
index 3e8bb5f..05f85a3 100644
--- a/src/HatsSignerGateBase.sol
+++ b/src/HatsSignerGateBase.sol
@@ -485,7 +485,7 @@ abstract contract HatsSignerGateBase is BaseGuard,
 ↪  SignatureDecoder, HatsOwnedIn
             safe.nonce() - 1 // view function
         );

-        uint256 validSigCount = countValidSignatures(txHash, signatures,
 ↪  signatures.length / 65);
+        uint256 validSigCount = countValidSignatures(txHash, signatures,
 ↪  safe.getThreshold());

         // revert if there aren't enough valid signatures
         if (validSigCount < safe.getThreshold() || validSigCount <
 ↪  minThreshold) {
```

Instead of checking all signatures, only the first `threshold` ones will be checked. Also there is no need to check the length of the `signatures` bytes. All those checks are done by the Safe already.

## Discussion

**spengrah**

Excellent find. The recommended fix makes sense.

**spengrah**

https://github.com/Hats-Protocol/hats-zodiac/pull/11

# Issue H-8: HatsSignerGate + MultiHatsSignerGate: more than maxSignatures can be claimed which leads to DOS in reconcileSignerCount

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/51

## Found by

roguereddwarf, cducrest-brainbot, GimelSec

## Summary

The `HatsSignerGate.claimSigner` and `MultiHatsSignerGate.claimSigner` functions allow users to become signers.

It is important that both functions do not allow that there exist more valid signers than `maxSigners`.

This is because if there are more valid signers than `maxSigners`, any call to `HatsSignerGateBase.reconcileSignerCount` reverts, which means that no transactions can be executed.

The only possibility to resolve this is for a valid signer to give up his signer hat. No signer will voluntarily give up his signer hat. And it is wrong that a signer must give it up. Valid signers that have claimed before `maxSigners` was reached should not be affected by someone trying to become a signer and exceeding `maxSigners`. In other words the situation where one of the signers needs to give up his signer hat should have never occurred in the first place.

## Vulnerability Detail

Think of the following scenario:

1. `maxSignatures=10` and there are 10 valid signers
2. The signers execute a transaction that calls `Safe.addOwnerWithThreshold` such that there are now 11 owners (still there are 10 valid signers)
3. One of the 10 signers is no longer a wearer of the hat and `reconcileSignerCount` is called. So there are now 9 valid signers and 11 owners
4. The signer that was no longer a wearer of the hat in the previous step now wears the hat again. However `reconcileSignerCount` is not called. So there are 11 owners and 10 valid signers. The HSG however still thinks there are 9 valid signers.

When a new signer now calls `claimSigner`, all checks will pass and he will be swapped for the owner that is not a valid signer:

```
// 9 >= 10 is false
if (currentSignerCount >= maxSigs) {
    revert MaxSignersReached();
}

// msg.sender is a new signer so he is not yet owner
if (safe.isOwner(msg.sender)) {
    revert SignerAlreadyClaimed(msg.sender);
}

// msg.sender is a valid signer, he wears the signer hat
if (!isValidSigner(msg.sender)) {
    revert NotSignerHatWearer(msg.sender);
}
```

So there are now 11 owners and 11 valid signers. This means when `reconcileSignerCount` is called, the following lines cause a revert:

```
function reconcileSignerCount() public {
    address[] memory owners = safe.getOwners();
    uint256 validSignerCount = _countValidSigners(owners);

    // 11 > 10
    if (validSignerCount > maxSigners) {
        revert MaxSignersReached();
    }
```

## Impact

As mentioned before, we end up in a situation where one of the valid signers has to give up his signer hat in order for the HSG to become operable again.

So one of the valid signers that has rightfully claimed his spot as a signer may lose his privilege to sign transactions.

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGate.sol#L36-L69

```
function claimSigner() public virtual {
    uint256 maxSigs = maxSigners; // save SLOADs
    uint256 currentSignerCount = signerCount;
```

```
    if (currentSignerCount >= maxSigs) {
        revert MaxSignersReached();
    }


    if (safe.isOwner(msg.sender)) {
        revert SignerAlreadyClaimed(msg.sender);
    }


    if (!isValidSigner(msg.sender)) {
        revert NotSignerHatWearer(msg.sender);
    }


    /*
    We check the safe owner count in case there are existing owners who are no
↪   longer valid signers.
    If we're already at maxSigners, we'll replace one of the invalid owners by
↪   swapping the signer.
    Otherwise, we'll simply add the new signer.
    */
    address[] memory owners = safe.getOwners();
    uint256 ownerCount = owners.length;


    if (ownerCount >= maxSigs) {
        bool swapped = _swapSigner(owners, ownerCount, maxSigs,
↪   currentSignerCount, msg.sender);
        if (!swapped) {
            // if there are no invalid owners, we can't add a new signer, so we
↪   revert
            revert NoInvalidSignersToReplace();
        }
    } else {
        _grantSigner(owners, currentSignerCount, msg.sender);
    }
}
```

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/MultiHatsSignerGate.sol#L41-L81

```
function claimSigner(uint256 _hatId) public {
    uint256 maxSigs = maxSigners; // save SLOADs
    uint256 currentSignerCount = signerCount;
```

```
    if (currentSignerCount >= maxSigs) {
        revert MaxSignersReached();
    }


    if (safe.isOwner(msg.sender)) {
        revert SignerAlreadyClaimed(msg.sender);
    }


    if (!isValidSignerHat(_hatId)) {
        revert InvalidSignerHat(_hatId);
    }


    if (!HATS.isWearerOfHat(msg.sender, _hatId)) {
        revert NotSignerHatWearer(msg.sender);
    }


    /*
    We check the safe owner count in case there are existing owners who are no
↪   longer valid signers.
    If we're already at maxSigners, we'll replace one of the invalid owners by
↪   swapping the signer.
    Otherwise, we'll simply add the new signer.
    */
    address[] memory owners = safe.getOwners();
    uint256 ownerCount = owners.length;


    if (ownerCount >= maxSigs) {
        bool swapped = _swapSigner(owners, ownerCount, maxSigs,
↪   currentSignerCount, msg.sender);
        if (!swapped) {
            // if there are no invalid owners, we can't add a new signer, so we
↪   revert
            revert NoInvalidSignersToReplace();
        }
    } else {
        _grantSigner(owners, currentSignerCount, msg.sender);
    }


    // register the hat used to claim. This will be the hat checked in
↪   `checkTransaction()` for this signer
    claimedSignerHats[msg.sender] = _hatId;
```

```
}
```

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L183-L217

```
function reconcileSignerCount() public {
    address[] memory owners = safe.getOwners();
    uint256 validSignerCount = _countValidSigners(owners);


    if (validSignerCount > maxSigners) {
        revert MaxSignersReached();
    }


    // update the signer count accordingly
    signerCount = validSignerCount;


    uint256 currentThreshold = safe.getThreshold();
    uint256 newThreshold;
    uint256 target = targetThreshold; // save SLOADs


    if (validSignerCount <= target && validSignerCount != currentThreshold) {
        newThreshold = validSignerCount;
    } else if (validSignerCount > target && currentThreshold < target) {
        newThreshold = target;
    }
    if (newThreshold > 0) {
        bytes memory data = abi.encodeWithSignature("changeThreshold(uint256)",
↪   validSignerCount);


        bool success = safe.execTransactionFromModule(
            address(safe), // to
            0, // value
            data, // data
            Enum.Operation.Call // operation
        );


        if (!success) {
            revert FailedExecChangeThreshold();
        }
    }
}
```

## Tool used

Manual Review

## Recommendation

The `HatsSignerGate.claimSigner` and `MultiHatsSignerGate.claimSigner` functions should call `reconcileSignerCount` such that they work with the correct amount of signers and the scenario described in this report cannot occur.

```
diff --git a/src/HatsSignerGate.sol b/src/HatsSignerGate.sol
index 7a02faa..949d390 100644
--- a/src/HatsSignerGate.sol
+++ b/src/HatsSignerGate.sol
@@ -34,6 +34,8 @@ contract HatsSignerGate is HatsSignerGateBase {
     /// @notice Function to become an owner on the safe if you are wearing the
 ↪  signers hat
     /// @dev Reverts if `maxSigners` has been reached, the caller is either
 ↪  invalid or has already claimed. Swaps caller with existing invalid owner if
 ↪  relevant.
     function claimSigner() public virtual {
+        reconcileSignerCount();
+
         uint256 maxSigs = maxSigners; // save SLOADs
         uint256 currentSignerCount = signerCount;
```

```
diff --git a/src/MultiHatsSignerGate.sol b/src/MultiHatsSignerGate.sol
index da74536..57041f6 100644
--- a/src/MultiHatsSignerGate.sol
+++ b/src/MultiHatsSignerGate.sol
@@ -39,6 +39,8 @@ contract MultiHatsSignerGate is HatsSignerGateBase {
     /// @dev Reverts if `maxSigners` has been reached, the caller is either
 ↪  invalid or has already claimed. Swaps caller with existing invalid owner if
 ↪  relevant.
     /// @param _hatId The hat id to claim signer rights for
     function claimSigner(uint256 _hatId) public {
+        reconcileSignerCount();
+
         uint256 maxSigs = maxSigners; // save SLOADs
         uint256 currentSignerCount = signerCount;
```

## Discussion

**spengrah**

I don't believe this is a duplicate of #46, which deals with number of owners being increased by another module, while the present issue deals with owners being

increased by the safe's signers. That means, however, that it be a duplicate of #118 and #170.

# Issue H-9: Signers can bypass checks and change threshold within a transaction

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/52

## Found by

obront, cducrest-brainbot

## Summary

The `checkAfterExecution()` function has checks to ensure that the safe's threshold isn't changed by a transaction executed by signers. However, the parameters used by the check can be changed midflight so that this crucial restriction is violated.

## Vulnerability Detail

The `checkAfterExecution()` is intended to uphold important invariants after each signer transaction is completed. This is intended to restrict certain dangerous signer behaviors. From the docs:

> /// @notice Post-flight check to prevent `safe` signers from removing this contract guard, changing any modules, or changing the threshold

However, the restriction that the signers cannot change the threshold can be violated.

To see how this is possible, let's check how this invariant is upheld. The following check is performed within the function:

```
if (safe.getThreshold() != _getCorrectThreshold()) {
    revert SignersCannotChangeThreshold();
}
```

If we look up `_getCorrectThreshold()`, we see the following:

```
function _getCorrectThreshold() internal view returns (uint256 _threshold) {
    uint256 count = _countValidSigners(safe.getOwners());
    uint256 min = minThreshold;
    uint256 max = targetThreshold;
    if (count < min) _threshold = min;
    else if (count > max) _threshold = max;
    else _threshold = count;
}
```

As we can see, this means that the safe's threshold after the transaction must equal the valid signers, bounded by the `minThreshold` and `maxThreshold`.

However, this check does not ensure that the value returned by `_getCorrectThreshold()` is the same before and after the transaction. As a result, as long as the number of owners is also changed in the transaction, the condition can be upheld.

To illustrate, let's look at an example:

- Before the transaction, there are 8 owners on the vault, all signers. targetThreshold == 10 and minThreshold == 2, so the safe's threshold is 8 and everything is good.
- The transaction calls `removeOwner()`, removing an owner from the safe and adjusting the threshold down to 7.
- After the transaction, there will be 7 owners on the vault, all signers, the safe's threshold will be 7, and the check will pass.

This simple example focuses on using `removeOwner()` once to decrease the threshold. However, it is also possible to use the safe's multicall functionality to call `removeOwner()` multiple times, changing the threshold more dramatically.

## Impact

Signers can change the threshold of the vault, giving themselves increased control over future transactions and breaking an important trust assumption of the protocol.

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L517-L519

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L533-L540

https://github.com/Hats-Protocol/safe-contracts/blob/c36bcab46578a442862d043e12a83fec41143dec/contracts/base/OwnerManager.sol#L70-L86

## Tool used

Manual Review

## Recommendation

Save the safe's current threshold in `checkTransaction()` before the transaction has executed, and compare the value after the transaction to that value from storage.



## Discussion

**spengrah**

In one sense, this is sort of ok since we're still always ensuring that the threshold is "correct" based on the number of valid signers on the safe, and changing the threshold doesn't allow . If a valid signer is removed by a safe tx, they can re-claim and then everything is back to the way it was.

But in another sense, it does make it fairly difficult to conceptualize the system, and it would also be ideal to successfully follow the documentation . In all, I think this is more of a medium severity.

The suggested solution should work, but likely isn't needed if we're also storing/comparing the full owner array in order to address #118 and #70.

cc @zobront

**spengrah**

Withdrawing the severity dispute since it would be quite painful if the signers continuously replaced existing signers to reach max signers, which could prevent replaced signers from reclaiming.

**spengrah**

https://github.com/Hats-Protocol/hats-zodiac/pull/5

**zobront**

@spengrah I believe this is still possible, if the safe is able to tamper with hat status or wearer eligibility and then reduce the threshold.

If this seems ok to you, then make sure to change the wording if your documentation and comments to not imply this isn't possible. If it's not ok with you, then I'd recommend implementing the recommended fix from this issue.

**spengrah**

@zobront ah ok, I see what you're saying. This is definitely something that should be documented more clearly: an assumption is that the safe itself (controlled by the signers) does not have authority over the `signersHat(s)`, ie is neither an admin, eligibility module, or toggle module. It wouldn't make sense for this to be the case and still expect the signers to not be able to control their own hat(s).

So, this is ok, and I'll push a documentation/comments update.

# Issue M-1: Hats.uri function can be DOSed by providing large details or imageURI string or cause large gas fees

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/2

## Found by

roguereddwarf

## Summary

The Hats.uri function returns a JSON representation of a hat's properties.

The issue is that the length of the `details` and `imageURI` string can be so big that the function consumes more than the maximum amount of gas. This would cause the transaction to revert.

Also a malicious user can just make the length so big that the transaction becomes very expensive. This would cause unnecessary expenses to anyone calling the function which is basically a loss of funds.

## Vulnerability Detail

The attack is only feasible on a low cost chain (so not on the ETH mainnet) because the attacker needs to set the large string which costs a lot of gas.

So let us consider Polygon which has a block gas limit of 30 million gas.

Furthermore I used this site to calculate the gas costs for the Polygon Network: https://www.cryptoneur.xyz/gas-fees-calculator

So in order to DOS the `Hats.uri` function we must make it consume `>30m` Gas.

It is not possible to set the `details` field to such a large string directly within one transaction because it would cost `>30m` Gas.

So we need to slowly increase the length of the `details` field using the `Hats.changeHatDetails` function.

Increasing the length by so much that the `Hats.uri` function consumes an extra `1m` Gas costs at most `15m` Gas (I tested this but obviously I cannot paste the code here because it would be ridiculously big).

At most we would have to spend `30 * 15m = 450m` Gas to execute the attack.

Using the calculator I linked, this costs `~92 USD` at the time of writing this.

This is within reach of an attacker. Also we must consider that gas prices may drop in the future.

## Impact

So the attacker can DOS the `Hats.uri` function or cause anyone calling it to spend up to `30m` Gas which is `~6 USD` on Polygon.

Also the attacker can make the hat immutable after performing the attack such that the string cannot be changed anymore.

## Code Snippet

https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/Hats.sol#L1072-L1138

https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/Hats.sol#L1221-L1223

## Tool used

Manual Review

## Recommendation

Introduce a reasonable limit for the length of the `details` and `imageURI` field.

## Discussion

**spengrah**

This would also DOS `viewHat`, which is currently the only way for other contracts to read hat status and several other properties. So I agree that a length limit would be appropriate.

Since it's not possible to set a long enough string in one transaction, I think the length checks only need to exist on the `changeHat...` functions.

TODO: determine appropriate length limit for `imageURI` and `details`, to be checked in `changeHatDetails` and `changeHatImageURI`.

**spengrah**

Tentative target cap is 7000 characters, which costs roughly 8,000,000 gas to write. cc @zobront

**spengrah**

https://github.com/Hats-Protocol/hats-protocol/pull/114

# Issue M-2: Hats can be overwritten

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/11

## Found by

carrot, Allarious

## Summary

Child hats can be created under a non-existent admin. Creating the admin allows overwriting the properties of the child-hats, which goes against the immutability of hats.

## Vulnerability Detail

When creating a hat, the code never checks if the admin passed actually exists or not. Thus it allows the creation of a hat under an admin who hasn't been created yet. Lets say top hat is 1.0.0, and we call admin the hat 1.1.0 and child is hat 1.1.1. The child can be created before admin. When admin (1.1.0) is created after this, it overwrites the `lastHatId` of the admin, as shown here https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/Hats.sol#L421-L439

```
function _createHat(
    uint256 _id,
    string calldata _details,
    uint32 _maxSupply,
    address _eligibility,
    address _toggle,
    bool _mutable,
    string calldata _imageURI
) internal returns (Hat memory hat) {
    hat.details = _details;
    hat.maxSupply = _maxSupply;
    hat.eligibility = _eligibility;
    hat.toggle = _toggle;
    hat.imageURI = _imageURI;
    hat.config = _mutable ? uint96(3 << 94) : uint96(1 << 95);
    _hats[_id] = hat;


    emit HatCreated(_id, _details, _maxSupply, _eligibility, _toggle, _mutable,
↪   _imageURI);
}
```

Now, the next eligible hat for this admin is 1.1.1, which is a hat that was already created and minted. This can allow the admin to change the properties of the child, even if the child hat was previously immutable. This contradicts the immutability of hats, and can be used to rug users in multiple ways, and is thus classified as high severity. This attack can be carried out by any hat wearer on their child tree, mutating their properties.

## Impact

## Code Snippet

The attack can be demonstrated with the following code which carries out the following steps:

1. Child 1.1.1 is created with max supply 10, and false mutability. Thus its properties should be locked.
2. Admin 1.1.0 is created
3. Child 1.1.1 is re-created, now with supply of 20, overwriting its previous instance
4. The children are shown to be on the same hatId, and their max supplies are shown to be different values.

```
function testATTACKoverwrite() public {
        vm.startPrank(address(topHatWearer));
        uint256 emptyAdmin = hats.getNextId(topHatId);
        uint256 child1 = hats.createHat(
            emptyAdmin,
            _details,
            10,
            _eligibility,
            _toggle,
            false,
            secondHatImageURI
        );
        (, uint256 maxsup, , , , , , ) = hats.viewHat(child1);
        assertEq(maxsup, 10);
        hats.createHat(
            topHatId,
            _details,
            _maxSupply,
            _eligibility,
            _toggle,
            false,
            secondHatImageURI
        );
        uint256 child2 = hats.createHat(
```

```
            emptyAdmin,
            _details,
            20,
            _eligibility,
            _toggle,
            false,
            secondHatImageURI
        );
        (,  maxsup, , , , , , , ) = hats.viewHat(child1);
        assertEq(child1, child2);
        assertEq(maxsup, 20);
    }
```

## Tool used

Manual Review

## Recommendation

Check if admin exists, before minting by checking any of its properties against default values

```
require(_hats[admin].maxSupply > 0, "Admin not created")
```

## Discussion

**spengrah**

The ability for an admin to skip levels when creating hats is a desired feature. However, we definitely do not want those hats to be able to overwritten if/when the skipped admins are created. Therefore, what we need to do is not overwrite a hat's `lastHatId` property when creating it.

For example, add something like the following to `_createHat`:

```
uint16 lastId = hat.lastHatId;
if (lastId > 0) hat.lastHatId = lastId;
```

**spengrah**

https://github.com/Hats-Protocol/hats-protocol/pull/109

# Issue M-3: Owners of linkedin tophats cannot have eligibility revoked

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/33

## Found by

obront

## Summary

Tophats that are linked to new admins are supposed to behave like normal hats, rather than tophats, but their `eligibility` variable will be permanently set to `address(0)`. This turns off the `badStanding` functionality, which allows all owners of the hat to be immune to punishments from their admins.

## Vulnerability Detail

When a Tophat is linked to a new admin, the assumption is that it will behave like any other immutable hat. From the docs:

> a tophat that has been linked (aka grafted) onto another hat tree is no longer considered a tophat

However, while most hats are not able to have their `toggle` or `eligibility` set to the zero address (which is carefully checked against in `createHat()`), tophats will always have them set to the zero address.

Since these hats are immutable, even when they are linked, there is no way to change the values of these two variables. They are permanently set to `address(0)`.

This is especially problematic for `eligibility`, as even the local value of `badStandings` that can be used if the event that there is no external contract is set in `_processHatWearerStatus()`, which can only be called if `eligibility` is either an EOA that calls `setHatWearerStatus()` or a contract that implements the `IHatsEligibility` interface.

This causes major problems for admins of linked tophats, as while they are expected to behave like other immutable hats, instead will not give the admin the privileges they need.

## Impact

Admins controlling linked tophats cannot turn off the eligibility of wearers who misbehave, giving tophat owners extra power and breaking the principle that linked tophats are no longer considered tophats.

## Code Snippet

https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/Hats.sol#L119-L127

## Tool used

Manual Review

## Recommendation

Create the ability for linked tophats to change their `toggle` and `eligibility` addresses one time upon being linked. This could be accomplished with an extra bit value in the `config` storage, which could be set and unset when tophats are linked and unlinked.

## Discussion

**spengrah**

The reason I didn't originally include something like this is that it could introduce weird behavior as trees are unlinked. If we allow eligibility and toggle modules to be added to linked tophats, then they'd need to be cleared if the tree is unlinked. But maybe that's actually not so bad.

We could even modify `approveLinkTopHatToTree` (and maybe `relinkTopHatToTree`?) to take toggle and eligibility parameters that (if non-zero) change a linked tophat's modules once on linking. I think this would forgo the need to write/read from a config bit.

When unlinked, those values would be reset to address(0).

Note: since `relinkTopHatToTree` can be called with the same origin and destination — ie to not move the topHat — under the above logic admins could use that function to change a linked topHat's eligibility and toggle an arbitrary number of times. I actually think this is fine, but something to note.

@zobront how does that sound?

**zobront**

@spengrah This sounds right to me. Part of linking and unlinking should be all the setup / wind down needed to turn it frmo tophat to normal hat and back. Your solution sound right.

# Issue M-4: Changing hat toggle address can lead to unexpected changes in status

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/34

## Found by

obront

## Summary

Changing the toggle address should not change the current status unless intended to. However, in the event that a contract's toggle status hasn't been synced to local state, this change can accidentally toggle the hat back on when it isn't intended.

## Vulnerability Detail

When an admin for a hat calls `changeHatToggle()`, the `toggle` address is updated to a new address they entered:

```
function changeHatToggle(uint256 _hatId, address _newToggle) external {
    if (_newToggle == address(0)) revert ZeroAddress();

    _checkAdmin(_hatId);
    Hat storage hat = _hats[_hatId];

    if (!_isMutable(hat)) {
        revert Immutable();
    }

    hat.toggle = _newToggle;

    emit HatToggleChanged(_hatId, _newToggle);
}
```

Toggle addresses can be either EOAs (who must call `setHatStatus()` to change the local config) or contracts (who must implement the `getHatStatus()` function and return the value).

The challenge comes if a hat has a toggle address that is a contract. The contract changes its toggle value to `false` but is never checked (which would push the update to the local state). The admin thus expects that the hat is turned off.

Then, the toggle is changed to an EOA. One would expect that, until a change is made, the hat would remain in the same state, but in this case, the hat defaults

back to its local storage state, which has not yet been updated and is therefore set to `true`.

Even in the event that the admin knows this and tries to immediately toggle the status back to `false`, it is possible for a malicious user to sandwich their transaction between the change to the EOA and the transaction to toggle the hat off, making use of a hat that should be off. This could have dramatic consequences when hats are used for purposes such as multisig signing.

## Impact

Hats may unexpectedly be toggled from `off` to `on` during toggle address transfer, reactivating hats that are intended to be turned off.

## Code Snippet

https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/Hats.sol#L623-L636

## Tool used

Manual Review

## Recommendation

The `changeHatToggle()` function needs to call `checkHatToggle()` before changing over to the new toggle address, to ensure that the latest status is synced up.

## Discussion

**spengrah**

The risk here is limited since toggle modules are treated as the source of truth for hat status and so admins should be aware that changing the toggle also changes the source of truth. But the recommended fix is simple and cheap enough that it makes sense to apply.

@zobront, does this same concept apply to eligibility?

**zobront**

@spengrah Agreed. The only situation I think would break a user's assumption is a change from contract => EOA, because in most cases the toggle will be saved in `hat.config` (so they won't need to take any action), but in this edge case it will be reset to an old version. That inconsistent behavior is the root of the problem.

The same is true with `eligibility`, but not sure if there's anything you can do about it, because it's on an individual hat by hat basis. If the `eligibility` contract is

changed, you can't go through all the hats and update them. So my advice is just to be sure to document it very clearly in this case.

**spengrah**

https://github.com/Hats-Protocol/hats-protocol/pull/116

**cducrest**

Escalate for 10 USDC

Disagree with being sever enough for being a medium. The Sherlock doc state "The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team." However, as was explained, this problem is present both for `status` and `eligibility`, and a fix is possible and implemented in https://github.com/Hats-Protocol/hats-protocol/pull/116 only for `status`, proving that the risk is acceptable for the `eligibility` part thus acceptable for the `status` part.

As noted in the comment by spengrah, the fix was implemented mostly because it is not expensive to do so, even though the risk is limited.

**sherlock-admin**

> Escalate for 10 USDC
>
> Disagree with being sever enough for being a medium. The Sherlock doc state "The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team." However, as was explained, this problem is present both for `status` and `eligibility`, and a fix is possible and implemented in https://github.com/Hats-Protocol/hats-protocol/pull/116 only for `status`, proving that the risk is acceptable for the `eligibility` part thus acceptable for the `status` part.
>
> As noted in the comment by spengrah, the fix was implemented mostly because it is not expensive to do so, even though the risk is limited.

You've created a valid escalation for 10 USDC!

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**

Escalation rejected

The issue is a valid medium Given the edge case that the attack is still possible and can cause a hat to be activated which should not be and vice versa.

**sherlock-admin**

> Escalation rejected

> The issue is a valid medium Given the edge case that the attack is still possible and can cause a hat to be activated which should not be and vice versa.

This issue's escalations have been rejected!

Watsons who escalated this issue will have their escalation amount deducted from their next payout.

# Issue M-5: targetThreshold can be set below minThreshold, violating important invariant

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/36

## Found by

Bauer, cducrest-brainbot, rvierdiiev, obront, carrot

## Summary

There are protections in place to ensure that `minThreshold` is not set above `targetThreshold`, because the result is that the max threshold on the safe would be less than the minimum required. However, this check is not performed when `targetThreshold` is set, which results in the same situation.

## Vulnerability Detail

When the `minThreshold` is set on `HatsSignerGateBase.sol`, it performs an important check that `minThreshold <= targetThreshold`:

```
function _setMinThreshold(uint256 _minThreshold) internal {
    if (_minThreshold > maxSigners || _minThreshold > targetThreshold) {
        revert InvalidMinThreshold();
    }

    minThreshold = _minThreshold;
}
```

However, when `targetThreshold` is set, there is no equivalent check that it remains above `minThreshold`:

```
function _setTargetThreshold(uint256 _targetThreshold) internal {
    if (_targetThreshold > maxSigners) {
        revert InvalidTargetThreshold();
    }

    targetThreshold = _targetThreshold;
}
```

This is a major problem, because if it is set lower than `minThreshold`, `reconcileSignerCount()` will set the safe's threshold to be this value, which is lower than the minimum, and will cause all tranasctions to fail.

## Impact

Settings that are intended to be guarded are not, which can lead to parameters being set in such a way that all transactions fail.

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L95-L114

## Tool used

Manual Review

## Recommendation

Perform a check in `_setTargetThreshold()` that it is greater than or equal to `minThreshold`:

```
function _setTargetThreshold(uint256 _targetThreshold) internal {
+   if (_targetThreshold < minThreshold) {
+     revert InvalidTargetThreshold();
+   }
    if (_targetThreshold > maxSigners) {
        revert InvalidTargetThreshold();
    }

    targetThreshold = _targetThreshold;
}
```

## Discussion

**spengrah**

https://github.com/Hats-Protocol/hats-zodiac/pull/8

# Issue M-6: Swap Signer fails if final owner is invalid due to off by one error in loop

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/38

## Found by

obront, james_wu

## Summary

New users attempting to call `claimSigner()` when there is already a full slate of owners are supposed to kick any invalid owners off the safe in order to swap in and take their place. However, the loop that checks this has an off-by-one error that misses checking the final owner.

## Vulnerability Detail

When `claimSigner()` is called, it adds the `msg.sender` as a signer, as long as there aren't already too many owners on the safe.

However, in the case that there are already the maximum number of owners on the safe, it performs a check whether any of them are invalid. If they are, it swaps out the invalid owner for the new owner.

```
if (ownerCount >= maxSigs) {
    bool swapped = _swapSigner(owners, ownerCount, maxSigs, currentSignerCount,
↪   msg.sender);
    if (!swapped) {
        // if there are no invalid owners, we can't add a new signer, so we
↪   revert
        revert NoInvalidSignersToReplace();
    }
}
```

```
function _swapSigner(
    address[] memory _owners,
    uint256 _ownerCount,
    uint256 _maxSigners,
    uint256 _currentSignerCount,
    address _signer
) internal returns (bool success) {
    address ownerToCheck;
    bytes memory data;

    for (uint256 i; i < _ownerCount - 1;) {
```

```
            ownerToCheck = _owners[i];

            if (!isValidSigner(ownerToCheck)) {
                // prep the swap
                data = abi.encodeWithSignature(
                    "swapOwner(address,address,address)",
                    _findPrevOwner(_owners, ownerToCheck), // prevOwner
                    ownerToCheck, // oldOwner
                    _signer // newOwner
                );

                // execute the swap, reverting if it fails for some reason
                success = safe.execTransactionFromModule(
                    address(safe), // to
                    0, // value
                    data, // data
                    Enum.Operation.Call // operation
                );

                if (!success) {
                    revert FailedExecRemoveSigner();
                }

                if (_currentSignerCount < _maxSigners) ++signerCount;
                break;
            }
            unchecked {
                ++i;
            }
        }
    }
```

This function is intended to iterate through all the owners, check if any is no longer valid, and — if that's the case — swap it for the new one.

However, in the case that all owners are valid except for the final one, it will miss the swap and reject the new owner.

This is because there is an off by one error in the loop, where it iterates through `for (uint256 i; i < _ownerCount - 1;)...`

This only iterates through all the owners up until the final one, and will miss the check for the validity and possible swap of the final owner.

## Impact

When only the final owner is invalid, new users will not be able to claim their role as signer, even through they should.

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGate.sol#L57-L69

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L308-L350

## Tool used

Manual Review

## Recommendation

Perform the loop with `ownerCount` instead of `ownerCount - 1` to check all owners:

```
- for (uint256 i; i < _ownerCount - 1;) {
+ for (uint256 i; i < _ownerCount ;) {
     ownerToCheck = _owners[i];
    ...
}
```

## Discussion

**spengrah**

https://github.com/Hats-Protocol/hats-zodiac/pull/12

# Issue M-7: If a hat is owned by address(0), phony signatures will be accepted by the safe

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/39

## Found by

obront, duc

## Summary

If a hat is sent to `address(0)`, the multisig will be fooled into accepting phony signatures on its behalf. This will throw off the proper accounting of signatures, allowing non-majority transactions to pass and potentially allowing users to steal funds.

## Vulnerability Detail

In order to validate that all signers of a transaction are valid signers, `HatsSignerGateBase.sol` implements the `countValidSignatures()` function, which recovers the signer for each signature and checks `isValidSigner()` on them.

The function uses `ecrecover` to get the signer. However, `ecrecover` is well known to return `address(0)` in the event that a phony signature is passed with a `v` value other than 27 or 28. See this example for how this can be done.

In the event that this is a base with only a single hat approved for signing, the `isValidSigner()` function will simply check if the owner is the wearer of a hat:

```
function isValidSigner(address _account) public view override returns (bool
↪   valid) {
    valid = HATS.isWearerOfHat(_account, signersHatId);
}
```

On the `Hats.sol` contract, this simply checks their balance:

```
function isWearerOfHat(address _user, uint256 _hatId) public view returns (bool
↪   isWearer) {
    isWearer = (balanceOf(_user, _hatId) > 0);
}
```

... which only checks if it is active or eligible...

```
function balanceOf(address _wearer, uint256 _hatId)
    public
    view
```

```
    override(ERC1155, IHats)
    returns (uint256 balance)
{
    Hat storage hat = _hats[_hatId];

    balance = 0;

    if (_isActive(hat, _hatId) && _isEligible(_wearer, hat, _hatId)) {
        balance = super.balanceOf(_wearer, _hatId);
    }
}
```

... which calls out to ERC1155, which just returns the value in storage (without any address(0) check)...

```
function balanceOf(address owner, uint256 id) public view virtual returns
↪   (uint256 balance) {
    balance = _balanceOf[owner][id];
}
```

The result is that, if a hat ends up owned by `address(0)` for any reason, this will give blanket permission for anyone to create a phony signature that will be accepted by the safe.

You could imagine a variety of situations where this may apply:

- An admin minting a mutable hat to address(0) to adjust the supply while waiting for a delegatee to send over their address to transfer the hat to
- An admin sending a hat to address(0) because there is some reason why they need the supply slightly inflated
- An admin accidentally sending a hat to address(0) to burn it

None of these examples are extremely likely, but there would be no reason for the admin to think they were putting their multisig at risk for doing so. However, the result would be a free signer on the multisig, which would have dramatic consequences.

## Impact

If a hat is sent to `address(0)`, any phony signature can be accepted by the safe, leading to transactions without sufficient support being executed.

This is particularly dangerous in a 2/3 situation, where this issue would be sufficient for a single party to perform arbitrary transactions.

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L547-L591

## Tool used

Manual Review

## Recommendation

The easiest option is to add a check in `countValidSignatures()` that confirms that `currentOwner != address(0)` after each iteration.

For extra security, you may consider implementing a check in `balanceOf()` that errors if we use `address(0)` as the address to check. (This is what OpenZeppelin does in their ERC721 implementation: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/7f028d69593342673492b0a0b1679e2a898cf1cf/contracts/token/ERC721/ERC721.sol#L62-L65)

## Discussion

**zobront**

No fix made for this but it's solved by the fix to #50.

# Issue M-8: If signer gate is deployed to safe with more than 5 existing modules, safe will be bricked

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/43

## Found by

obront, juancito, roguereddwarf, cducrest-brainbot

## Summary

`HatsSignerGate` can be deployed with a fresh safe or connected to an existing safe. In the event that it is connected to an existing safe, it pulls the first 5 modules from that safe to count the number of connected modules. If there are more than 5 modules, it silently only takes the first five. This results in a mismatch between the real number of modules and `enabledModuleCount`, which causes all future transactions to revert.

## Vulnerability Detail

When a `HatsSignerGate` is deployed to an existing safe, it pulls the existing modules with the following code:

```
(address[] memory modules,) =
↪  GnosisSafe(payable(_safe)).getModulesPaginated(SENTINEL_MODULES, 5);
uint256 existingModuleCount = modules.length;
```

Because the modules are requested paginated with `5` as the second argument, it will return a maximum of 5 modules. If the safe already has more than 5 modules, only the first 5 will be returned.

The result is that, while the safe has more than 5 modules, the gate will be set up with `enabledModuleCount = 5 + 1`.

When a transaction is executed, `checkTransaction()` will get the hash of the first 6 modules:

```
(address[] memory modules,) = safe.getModulesPaginated(SENTINEL_OWNERS,
↪  enabledModuleCount);
_existingModulesHash = keccak256(abi.encode(modules));
```

After the transaction, the first 7 modules will be checked to compare it:

```
(address[] memory modules,) = safe.getModulesPaginated(SENTINEL_OWNERS,
↪  enabledModuleCount + 1);
if (keccak256(abi.encode(modules)) != _existingModulesHash) {
```

```
    revert SignersCannotChangeModules();
}
```

Since it already had more than 5 modules (now 6, with HatsSignerGate added), there will be a 7th module and the two hashes will be different. This will cause a revert.

This would be a high severity issue, except that in the comments for the function it says:

> /// @dev Do not attach HatsSignerGate to a Safe with more than 5 existing modules; its signers will not be able to execute any transactions

This is the correct recommendation, but given the substantial consequences of getting it wrong, it should be enforced in code so that a safe with more modules reverts, rather than merely suggested in the comments.

## Impact

If a HatsSignerGate is deployed and connected to a safe with more than 5 existing modules, all future transactions sent through that safe will revert.

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateFactory.sol#L124-L141

## Tool used

Manual Review

## Recommendation

The `deployHatsSignerGate()` function should revert if attached to a safe with more than 5 modules:

```
function deployHatsSignerGate(
    uint256 _ownerHatId,
    uint256 _signersHatId,
    address _safe, // existing Gnosis Safe that the signers will join
    uint256 _minThreshold,
    uint256 _targetThreshold,
    uint256 _maxSigners
) public returns (address hsg) {
    // count up the existing modules on the safe
    (address[] memory modules,) =
↪   GnosisSafe(payable(_safe)).getModulesPaginated(SENTINEL_MODULES, 5);
```

```
    uint256 existingModuleCount = modules.length;
+   (address[] memory modulesWithSix,) =
 ↪  GnosisSafe(payable(_safe)).getModulesPaginated(SENTINEL_MODULES, 6);
+   if (modules.length != moduleWithSix.length) revert TooManyModules();

    return _deployHatsSignerGate(
        _ownerHatId, _signersHatId, _safe, _minThreshold, _targetThreshold,
 ↪  _maxSigners, existingModuleCount
    );
}
```

## Discussion

**spengrah**

Based on other findings, considering removing the ability for any other modules to exist alongside HSG. If we go that route, the fix here would likely be to revert if `modulesWith1.length > 0`.

cc @zobront

**zobront**

Great, I agree with that.

**spengrah**

https://github.com/Hats-Protocol/hats-zodiac/pull/10

# Issue M-9: Safe threshold can be set above target threshold, causing transactions to revert

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/44

## Found by

cducrest-brainbot, cccz, duc, unforgiven, obront, Allarious

## Summary

If a `targetThreshold` is set below the safe's threshold, the `reconcileSignerCount()` function will fail to adjust the safe's threshold as it should, leading to a mismatch that causes all transactions to revert.

## Vulnerability Detail

It is possible and expected that the `targetThreshold` can be lowered, sometimes even lower than the current safe threshold.

In the `setTargetThreshold()` function, there is an automatic update to lower the safe threshold accordingly. However, in the event that the `signerCount < 2`, it will not occur. This could easily happen if, for example, the hat is temporarily toggled off.

But this should be fine! In this instance, when a new transaction is processed, `checkTransaction()` will be called, which calls `reconcileSignerCount()`. This should fix the problem by resetting the safe's threshold to be within the range of `minThreshold` to `targetThreshold`.

However, the logic to perform this update is faulty.

```
uint256 currentThreshold = safe.getThreshold();
uint256 newThreshold;
uint256 target = targetThreshold; // save SLOADs

if (validSignerCount <= target && validSignerCount != currentThreshold) {
    newThreshold = validSignerCount;
} else if (validSignerCount > target && currentThreshold < target) {
    newThreshold = target;
}
if (newThreshold > 0) { ... update safe threshold ... }
```

As you can see, in the event that the `validSignerCount` is lower than the target threshold, we update the safe's threshold to `validSignerCount`. That is great.

In the event that `validSignerCount` is greater than threshold, we should be setting the safe's threshold to `targetThreshold`. However, this only happens in the `else if`

clause, when `currentThreshold < target`.

As a result, in the situation where `target < current <= validSignerCount`, we will leave the current safe threshold as it is and not lower it. This results in a safe threshold that is greater than `targetThreshold`.

Here is a simple example:

- valid signers, target threshold, and safe's threshold are all 10
- the hat is toggled off
- we lower target threshold to 9
- the hat is toggled back on
- `if` block above (`validSignerCount <= target && validSignerCount != currentThreshold`) fails because `validSignerCount > target`
- `else if` block above (`validSignerCount > target && currentThreshold < target`) fails because `currentThreshold > target`
- as a result, `newThreshold == 0` and the safe isn't updated
- the safe's threshold remains at 10, which is greater than target threshold

In the `checkAfterExecution()` function that is run after each transaction, there is a check that the threshold is valid:

```
if (safe.getThreshold() != _getCorrectThreshold()) {
    revert SignersCannotChangeThreshold();
}
```

The `_getCorrectThreshold()` function checks if the threshold is equal to the valid signer count, bounded by the `minThreshold` on the lower end, and the `targetThreshold` on the upper end:

```
function _getCorrectThreshold() internal view returns (uint256 _threshold) {
    uint256 count = _countValidSigners(safe.getOwners());
    uint256 min = minThreshold;
    uint256 max = targetThreshold;
    if (count < min) _threshold = min;
    else if (count > max) _threshold = max;
    else _threshold = count;
}
```

Since our threshold is greater than `targetThreshold` this check will fail and all transactions will revert.

## Impact

A simple change to the `targetThreshold` fails to propagate through to the safe's threshold, which causes all transactions to revert.

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L95-L114

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L183-L217

## Tool used

Manual Review

## Recommendation

Edit the if statement in `reconcileSignerCount()` to always lower to the `targetThreshold` if it exceeds it:

```
-if (validSignerCount <= target && validSignerCount != currentThreshold) {
+if (validSignerCount <= target) {
    newThreshold = validSignerCount;
-} else if (validSignerCount > target && currentThreshold < target) {
+} else {
    newThreshold = target;
}
-if (newThreshold > 0) { ... update safe threshold ... }
+if (newThreshold != currentThreshold) { ... update safe threshold ... }
```

## Discussion

**spengrah**

This issue is also resolved by using a dynamic signerCount() function instead of relying on the signerCount state var. This is implemented in https://github.com/Hats-Protocol/hats-zodiac/pull/6, with an additional test to demonstrate added in https://github.com/Hats-Protocol/hats-zodiac/pull/7

# Issue M-10: Can get around hats per level constraints using phantom levels

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/47

## Found by

obront, 0xMojito

## Summary

Each hat has the ability to mint `2^16 = 65,536` direct children. The id of the next child to be minted is saved in `_hats[_admin].lastHatId` and is incremented with each new child created. Once it hits 65,536, no more children can be created, as specified in the comments:

> // use the overflow check to constrain to correct number of hats per level

However, this check can be bypassed by using phantom levels. The basic idea is that there is nothing stopping an admin from creating a new hat and setting the admin to a non-existent path, using intermediate hatIds of 0 to accomplish this.

## Vulnerability Detail

Here's an example to make it more concrete:

- I am a topHat with domain 0x00000001
- I have created all 65,536 possible children (0x1.0001 to 0x1.ffff)
- But nothing is stopping me from setting 0x1.0000.0001 as the admin and continuing to create hats
- This hatId cannot possible exist (because no hatId immediately following a `0000` can actually be minted)
- Therefore, it is functionally the same as having 0x1 as the only admin
- This user can then continue to create children from 0x1.0000.0001.0001 to 0x1.0000.0001.ffff
- This can be repeated at each level, creating the ability to have unlimited direct children

## Impact

The constraints around the number of direct descendants that an individual hat can have can be easily violated.

## Code Snippet

https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/Hats.sol#L143-L170

## Tool used

Manual Review

## Recommendation

Add logic in `isAdminOfHat()` to ensure that each hat along the path being checked has a `maxSupply > 0` and is therefore a real hat.

## Discussion

**spengrah**

The child count constraint only exists because we have to be able to represent the hat ids correctly within the available 256 bits.

Assuming we appropriately mitigate the over-writing risk described by #11 and #72, phantom levels is actually kind of a nice feature to allow a given hat to have more than `2**16-1` immediate children. After all, they can achieve something close to this anyways by creating grandchild hats with a non-existent intermediate admin of 0001.

**spengrah**

Based on the impact of this issue described in #64, I'm going to remove the dispute of this one. We should ensure that all hat ids are valid hat ids (ie have no empty levels) to facilitate interactions and interpretation by other protocols, indexers, and apps.

TODO devise an efficient method for ensuring that the admin parameter of `createHat`, `requestLinkTopHatToTree`, and `relinkTopHatWithinTree` do not have empty levels.

**spengrah**

@zobront, my initial idea is to create a new pure function that traverses up the admin's id and returns false if it finds an empty level. We'd call that function from within the above 3 functions and revert if false. Should be relatively cheap since its just bitwise ops.

```
function isValidHatId(uint256 _hatId) public pure returns (bool validHatId) {
  // valid top hats are valid hats
  if (isLocalTopHat(_hatId)) return true;
  uint32 level = getLocalHatLevel(_hatId);
```

```
  uint256 admin;
  // for each subsequent level up the tree, check if the level is 0 and return
↪  false if so
  for (level; level > 0; ++level) {
    // find the admin at this level; we don't need to check _hatId itself since
↪  the getAdmin call already ignores trailing empty levels
    admin = getAdminAtLocalLevel(_hatId, level);
    // truncate to the 16 bits relevant to the direct admin and return false if 0
    if (uint16(admin) == 0) return false;
    }
  // if there are no empty levels, return true
  return true;
}
```

**spengrah**

I do think its worth considering whether we actually *do* want to support empty
↪  levels. Quick pros/cons list:

| Pros | Cons |
| --- | --- |
|  More flexibility for admins to create > 2**32 direct child hats  |  Doesn't
↪  work for admins at level 1 |
|  Increases the maximum number of hats per tree  |  Potential for confusion by
↪  users, protocols, indexers, and apps; would need to document very clearly  |
|  Not requiring the isValidHatId in `createHat` check saves a small amount of
↪  gas  |  More complexity, in general  |

Interested in what @nintynick @davehrlichman think here.

**spengrah**

Decision: in the spirit of keeping the protocol as simple as possible, we're
↪  going to close this loophole ~~/feature~~ with the above `isValidHatId`
↪  check.

cc @zobront

**zobront**

@spengrah Instead of adding an additional check, couldn't this be a part of the
↪  the `_adminCheck()`, since you're already traversing the tree to check `if
↪  (isWearerOfHat(_user, getAdminAtLocalLevel(_hatId, adminLocalHatLevel))) {
↪  ...`.

There may be a situation where this causes a problem. I can dig in deeper, but
↪  wanted to float the idea.

```
On the whole, agreed with the solution.

**spengrah**

@zobront my thinking is that I'd prefer adding cost to `createHat` over
↪  `isAdminOfHat` since the former is called just once while the latter is
↪  called many times over the life of the hat.

**zobront**

@spengrah Very good point, you're right.

**spengrah**

https://github.com/Hats-Protocol/hats-protocol/pull/108



# Issue M-11: [Medium][Outdated State] `_removeSigner` incorrectly updates
↪  `signerCount` and safe `threshold`

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/79

## Found by
Bauer, Allarious, duc

## Summary
`_removeSigner` can be called whenever a signer is no longer valid to remove an
↪  invalid signer. However, under certain situations, `removeSigner`
↪  incorrectly reduces the number of `signerCount` and sets the `threshold`
↪  incorrectly.

## Vulnerability Detail
`_removeSigner` uses the code snippet below to decide if the number of
↪  `signerCount` should be reduced:
```solidity
        if (validSignerCount == currentSignerCount) {
            newSignerCount = currentSignerCount;
        } else {
            newSignerCount = currentSignerCount - 1;
        }
```
```

If first clause is supposed to be activated when `validSignerCount` and `currentSignerCount` are still in sync, and we want to remove an invalid signer. The second clause is for when we need to identify a previously active signer which is inactive now and want to remove it. However, it does not take into account if a previously in-active signer became active. In the scenario described below, the

`signerCount` would be updated incorrectly:

(1) Lets imagine there are 5 signers where 0, 1 and 2 are active while 3 and 4 are inactive, the current `signerCount = 3` (2) In case number 3 regains its hat, it will become active again (3) If we want to delete signer 4 from the owners' list, the `_removeSigner` function will go through the signers and find 4 valid signers, since there were previously 3 signers, `validSignerCount == currentSignerCount` would be false. (4) In this case, while the number of `validSingerCount` increased, the `_removeSigner` reduces one.

## Impact

This can make the `signerCount` and safe `threshold` to update incorrectly which can cause further problems, such as incorrect number of signatures needed.

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L387

## Code Snippet

No code snippet provided

## Tool used

Manual Review

## Recommendation

Check if the number of `validSignerCount` decreased instead of checking equality:

```
@line 387 HatsSignerGateBase
- if (validSignerCount == currentSignerCount) {
+ if (validSignerCount >= currentSignerCount) {
```

## Discussion

**spengrah**

This is another issue that can be avoided by using a dynamic `getSignerCount` instead of relying on the `signerCount` state variable.

cc @zobront

**spengrah**

https://github.com/Hats-Protocol/hats-zodiac/pull/6

# Issue M-12: The Hats contract needs to override the ERC1155.balanceOfBatch function

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/85

## Found by

roguereddwarf, ktg, cccz, GimelSec

## Summary

The Hats contract does not override the ERC1155.balanceOfBatch function

## Vulnerability Detail

The Hats contract overrides the ERC1155.balanceOf function to return a balance of 0 when the hat is inactive or the wearer is ineligible.

```
function balanceOf(address _wearer, uint256 _hatId)
    public
    view
    override(ERC1155, IHats)
    returns (uint256 balance)
{
    Hat storage hat = _hats[_hatId];

    balance = 0;

    if (_isActive(hat, _hatId) && _isEligible(_wearer, hat, _hatId)) {
        balance = super.balanceOf(_wearer, _hatId);
    }
}
```

But the Hats contract does not override the ERC1155.balanceOfBatch function, which causes balanceOfBatch to return the actual balance no matter what the circumstances.

```
function balanceOfBatch(address[] calldata owners, uint256[] calldata ids)
    public
    view
    virtual
    returns (uint256[] memory balances)
{
    require(owners.length == ids.length, "LENGTH_MISMATCH");

    balances = new uint256[](owners.length);
```

```
        // Unchecked because the only math done is incrementing
        // the array index counter which cannot possibly overflow.
        unchecked {
            for (uint256 i = 0; i < owners.length; ++i) {
                balances[i] = _balanceOf[owners[i]][ids[i]];
            }
        }
    }
```

## Impact

This will make balanceOfBatch return a different result than balanceOf, which may cause errors when integrating with other projects

## Code Snippet

https://github.com/Hats-Protocol/hats-protocol/blob/main/src/Hats.sol#L1149-L1162 https://github.com/Hats-Protocol/hats-protocol/blob/main/lib/ERC1155/ERC1155.sol#L118-L139

## Tool used

Manual Review

## Recommendation

Consider overriding the ERC1155.balanceOfBatch function in Hats contract to return 0 when the hat is inactive or the wearer is ineligible.

## Discussion

**spengrah**

https://github.com/Hats-Protocol/hats-protocol/pull/102

**zobront**

Reverting solves the problem, but I'm curious why revert instead of just replacing:

```
balances[i] = _balanceOf[owners[i]][ids[i]];
```

with...

```
balances[i] = balanceOf(owners[i], ids[i]);
```

**zobront**

Reverting solves the problem, but I'm curious why revert instead of just replacing:

```
balances[i] = _balanceOf[owners[i]][ids[i]];
```

with...

```
balances[i] = balanceOf(owners[i], ids[i]);
```

# Issue M-13: Usage of HSG for existing safe can brick safe

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/93

## Found by

cducrest-brainbot

## Summary

The `HatsSignerGateFactory` allows for deployment of HSG / MHSG for existing safes. I believe the intention is to let the user call `enableModule()` and `setGuard()` on the safe after deployme with the HSG / MHSG address.

This can result in unmatching values of `maxSigners` in the HSG and number of valid signers of the safe. That will prevent further interaction with the safe rendering it unusable.

## Vulnerability Detail

If a safe has 10 owners with valid hats, and a HSG / MHSG is deployed with a value of `maxSigners < 10` and this HSG / MHSG is wired to the safe, the checks for `validSignerCount <= maxSigners` will revert in the HSG.

These checks are present in `reconcileSignerCount` and `claimSigner`. However `reconcileSignerCount` is a core function that is called by `checkTransaction()`, the pre-flight check on the safe transaction.

## Impact

The safe will not be able to execute any transaction until the number of valid signers is lowered (some hat wearers give up their hats / some hats turns invalid ...)

## Code Snippet

`reconcileSignerCount` checks maxSigners value:

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L183-L189

It is called during `checkTransaction`:

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L464

The value is set once during setup and not changeable:

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L66-L84

## Tool used

Manual Review

## Recommendation

Allow this value to be changed by owner, or have a function that checks the HSG is safe before making it active after it is wired to an existing safe.

## Discussion

**spengrah**

I like the recommendation to add a function to check that HSG is can be safely wired up to an existing Safe.

Another approach could be to check in the factory that `maxSigners >= hat.currentSupply` — or use the recommendation from #104 of ensuring that `maxSigners > safe.getOwners()` — but this wouldn't completely solve it because more hats could be minted to existing signers between deployment and wiring up.

cc @zobront

**zobront**

@spengrah I agree that the check needs to be at the "wiring up" time and not earlier.

**spengrah**

Re-adding the severity dispute tag since existing signers (who have full control over the safe prior to attaching HSG) would be both the ones attaching HSG in the first place and needing to renounce or be revoked if they made a mistake to attach. To me, this means that this is not a high severity bug, since attaching / fixing an attachment of HSG are incentive-aligned.

# Issue M-14: Unbound recursive function call can use unlimited gas and break hats operation

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/96

## Found by

Ace-30, unforgiven, chaduke, 0xMojito, Allarious, GimelSec

## Summary

some of the functions in the Hats and HatsIdUtilities contracts has recursive logics without limiting the number of iteration, this can cause unlimited gas usage if hat trees has huge depth and it won't be possible to call the contracts functions. functions `getImageURIForHat()`, `isAdminOfHat()`, `getTippyTopHatDomain()` and `noCircularLinkage()` would revert and because most of the logics callings those functions so contract would be in broken state for those hats.

## Vulnerability Detail

This is function `isAdminOfHat()` code:

```
function isAdminOfHat(address _user, uint256 _hatId) public view returns (bool
↪  isAdmin) {
    uint256 linkedTreeAdmin;
    uint32 adminLocalHatLevel;
    if (isLocalTopHat(_hatId)) {
        linkedTreeAdmin = linkedTreeAdmins[getTopHatDomain(_hatId)];
        if (linkedTreeAdmin == 0) {
            // tree is not linked
            return isAdmin = isWearerOfHat(_user, _hatId);
        } else {
            // tree is linked
            if (isWearerOfHat(_user, linkedTreeAdmin)) {
                return isAdmin = true;
            } // user wears the treeAdmin
            else {
                adminLocalHatLevel = getLocalHatLevel(linkedTreeAdmin);
                _hatId = linkedTreeAdmin;
            }
        }
    } else {
        // if we get here, _hatId is not a tophat of any kind
        // get the local tree level of _hatId's admin
        adminLocalHatLevel = getLocalHatLevel(_hatId) - 1;
    }
```

```
    // search up _hatId's local address space for an admin hat that the _user
↪   wears
    while (adminLocalHatLevel > 0) {
        if (isWearerOfHat(_user, getAdminAtLocalLevel(_hatId,
↪   adminLocalHatLevel))) {
            return isAdmin = true;
        }
        // should not underflow given stopping condition > 0
        unchecked {
            --adminLocalHatLevel;
        }
    }

    // if we get here, we've reached the top of _hatId's local tree, ie the
↪   local tophat
    // check if the user wears the local tophat
    if (isWearerOfHat(_user, getAdminAtLocalLevel(_hatId, 0))) return isAdmin =
↪   true;

    // if not, we check if it's linked to another tree
    linkedTreeAdmin = linkedTreeAdmins[getTopHatDomain(_hatId)];
    if (linkedTreeAdmin == 0) {
        // tree is not linked
        // we've already learned that user doesn't wear the local tophat, so
↪   there's nothing else to check; we return false
        return isAdmin = false;
    } else {
        // tree is linked
        // check if user is wearer of linkedTreeAdmin
        if (isWearerOfHat(_user, linkedTreeAdmin)) return true;
        // if not, recurse to traverse the parent tree for a hat that the user
↪   wears
        isAdmin = isAdminOfHat(_user, linkedTreeAdmin);
    }
}
```

As you can see this function calls itself recursively to check that if user is wearer of the one of the upper link hats of the hat or not. if the chain(depth) of the hats in the tree become very long then this function would revert because of the gas usage and the gas usage would be high enough so it won't be possible to call this function in a transaction. functions `getImageURIForHat()`, `getTippyTopHatDomain()` and `noCircularLinkage()` has similar issues and the gas usage is depend on the tree depth. the issue can happen suddenly for hats if the top level topHat decide to add link, for example:

1. Hat1 is linked to chain of the hats that has 1000 "root hat" and the topHat



(tippy hat) is TIPHat1.

2. Hat2 is linked to chain of the hats that has 1000 "root hat" and the topHat (tippy hat) is TIPHat2.
3. admin of the TIPHat1 decides to link it to the Hat2 and all and after performing that the total depth of the tree would increase to 2000 and transactions would cost double time gas.

## Impact

it won't be possible to perform actions for those hats and funds can be lost because of it.

## Code Snippet

https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/Hats.sol#L831-L883

https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/Hats.sol#L1022-L1067

https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/HatsIdUtilities.sol#L194-L200

https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/HatsIdUtilities.sol#L184-L188

## Tool used

Manual Review

## Recommendation

code should check and make sure that hat levels has a maximum level and doesn't allow actions when this level breaches. (keep depth of each tophat's tree and update it when actions happens and won't allow actions if they increase depth higher than the threshold)

## Discussion

**spengrah**

Tentative solution is to cap the number of nested trees at a reasonable value that ensures the tippy top hat will always be able to exercise its admin rights over the lowest tree, eg 10

1. Count the number of nested trees in the would-be child tree

2. Count the number of nested trees in the would-be parent tree
3. If sum of values (1) and (2) exceeds the cap, revert

We can calculate (1) and (2) by incrementing a counter each step up as we traverse the `linkedTreeAdmins` mapping, potentially in `noCircularLinkage()`.

cc @zobront

**spengrah**

Will need to test out gas impacts of higher numbers of nested trees to find a good cap value.

**zobront**

@spengrah Agree this is the best solution. Two questions:

- Will you try to cap to an amount where the gas price will be reasonable, or where it'll fit in a block?
- If this differs by chain, will you have different versions by chain, or keep them consistent at mainnet values?

**spengrah**

Great questions @zobront. Overall, I would strongly prefer to not have different versions across different networks, so will likely constrain cheaper / higher capacity networks based on the limits of more expensive networks.

That said, I think it should be up to the user org to determine the cost their willing to spend to add more trees, so overall I think I'm likely to set the cap based on block size rather than gas price.

**spengrah**

@zobront after further research, it's actually going to be pretty expensive to enforce a cap. We can relatively easily count the depth of the would-be parent tree (step 2 above) since we can traverse up the "linked list" of the linkedTreeAdmin mapping. But counting the depth of the would-be child would require a new data model since we need to count *down*.

So, at this point in the process, what I think is most appropriate is to document this risk and recommend that any tophats that have more than N (say, 10) nested trees below them not grant any authorities to the trees below N.

**spengrah**

@zobront We figured out a fix!

Was thinking that the additional data structure needed to count down would be onerous, but realized that we can just add another mapping that goes the other way, basically creating a doubly-linked list without having to impact the first mapping. So we can now count down pretty easily.

https://github.com/Hats-Protocol/hats-protocol/pull/118

# Issue M-15: middle level admins can steal child trees because function unlinkTopHatFromTree() is callable by them

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/116

## Found by

unforgiven

## Summary

in normal scenario middle level admins can't relink a child tree to another tree and when code link a tree it checks that the old and new admin is in the same tree(have same tippy hat). but middle level admins can bypass this buy using unlink logic.

## Vulnerability Detail

a middle level admin can perform this actions: (x -> y) means x is admin of y (y linked to x)

1. suppose we have this tree: Root0 (admin0) -> H1 (Admin1) -> H2 (Admin2) -> Root3 (Admin3) -> H4 (Admin4). (H1, H2, H4 are hats and Root0 is tophat and Root3 is "root tree")
2. now Admin2 wants to remove (Root3 -> H4) tree from higher level admins(Admin1) access.
3. Admin2 would create new TopHat Root1 and link it to H2. (H2 -> Root1).
4. now Admin2 would change the link of Root3 from H2 to Root1 by calling `relinkTopHatWithinTree(Root3, Root1)`. because Admin2 is admins of the H2 and both Root3 and Root1 is linked to the H2 so this action would be possible. the tree would become: (Root2 -> H1 -> H2 -> Root1 -> Root3 -> H4)
5. now Admin2 would call `unlinkTopHatFromTree(Root3)` and unlink Root1 from Root0 tree. because Admin2 created Root1 he would become admin of the Root1 and would be only one that have admin access to Root3. (Root1 -> Root3 -> H4)

simple unlinking can't always make middle level admin to be new admin so middle level admin should perform one relink and then unlink.

## Impact

middle level admin can steal the child trees and become the solo admin of them

## Code Snippet

https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/Hats.sol#L726-L732
https://github.com/Hats-Protocol/hats-protocol/blob/fafcfdf046c0369c1f9e077eacd94a328f9d7af0/src/Hats.sol#L739-L755

## Tool used

Manual Review

## Recommendation

This is a logical issue and fixing it is a little hard. one fix is that only allow the tippy hat admin to unlink a child tree. one another solution is only allow relink if the child tree to upper level trees not another sub level tree. (the path from the tree to the tippy hat)

## Discussion

**spengrah**

This is a good one, and a tough one! I can see a few potential resolutions (including those recommended by op):

1. Only allow tippy top hats to relink child trees
2. Only allow tippy top hats to unlink child trees
3. Only allow relinking if the destination is the main tree
4. Remove relinking altogether and require that relinking be done in three steps: a) admin unlinks, b) unlinked top hat requests new link, c) destination admin approves new link

Not yet sure which is best...will thinking more about this.

**spengrah**

Likely going with option 3, cc @zobront

**zobront**

@spengrah I agree with this. Best solution to me seems to be continuing to allow anyone to relink but checking that tippy top hat of sending tree and receiving tree are the same. Is that what you have in mind to enforce option 3?

**spengrah**

> I agree with this. Best solution to me seems to be continuing to allow anyone to relink but checking that tippy top hat of sending tree and

> receiving tree are the same. Is that what you have in mind to enforce option 3?

@zobront actually that is not quite what I was thinking, but it's better, so now its what I'm thinking :)

**spengrah**

> checking that tippy top hat of sending tree and receiving tree are the same

Actually, this is not sufficient since the described attack begins under the same tippy top hat. I think what we need to do instead is check that one or more of the following is true:

1) destination is the same local tree as origin, ie by checking that they both share the same local tophat
2) destination is the tippy top hat's local tree, ie by checking that the tippy tophat is equal to the local tophat for the destination
3) caller wears the tippyTopHat

cc @zobront

**spengrah**

https://github.com/Hats-Protocol/hats-protocol/pull/113

**zobront**

Escalate for 10 USDC

This is a valid issue but the severity is wrong.

A core assumption of the protocol is that admins are trustable. If we start questioning the admin trust assumptions, a whole new field of bugs opens up.

With that in mind, it does feel like users would assume that if they are higher up the tree, they wouldn't be able to lose control of a sub hat. While they are giving complete control of the subhat to the mid-level admin, it seems like a fair assumption that that user wouldn't be able to remove their ownership over the hat.

For that reason, I think it's fair to call this a Medium, but it's definitely not a High, as the only possible paths for exploit are through a trusted address.

**sherlock-admin**

> Escalate for 10 USDC
>
> This is a valid issue but the severity is wrong.
>
> A core assumption of the protocol is that admins are trustable. If we start questioning the admin trust assumptions, a whole new field of bugs opens up.

> With that in mind, it does feel like users would assume that if they are higher up the tree, they wouldn't be able to lose control of a sub hat. While they are giving complete control of the subhat to the mid-level admin, it seems like a fair assumption that that user wouldn't be able to remove their ownership over the hat.
>
> For that reason, I think it's fair to call this a Medium, but it's definitely not a High, as the only possible paths for exploit are through a trusted address.

You've created a valid escalation for 10 USDC!

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**

Escalation accepted

Given the complexity around the user admin structure and the permissions, and the preconditions for this attack to happen, considering this issue a valid medium.

**sherlock-admin**

> Escalation accepted
>
> Given the complexity around the user admin structure and the permissions, and the preconditions for this attack to happen, considering this issue a valid medium.

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

**sherlock-admin**

> Escalation accepted
>
> Given the complexity around the user admin structure and the permissions, and the preconditions for this attack to happen, considering this issue a valid medium.

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

# Issue M-16: Owners can be swapped even though they still wear their signer hats

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/118

## Found by

minhtrng, Dug

## Summary

`HatsSignerGateBase` does not check for a change of owners post-flight. This allows a group of actors to collude and replace opposing signers with cooperating signers, even though the replaced signers still wear their signer hats.

## Vulnerability Detail

The `HatsSignerGateBase` performs various checks to prevent a multisig transaction to tamper with certain variables. Something that is currently not checked for in `checkAfterExecution` is a change of owners. A colluding group of malicious signers could abuse this to perform swaps of safe owners by using a delegate call to a corresponding malicious contract. This would bypass the requirement of only being able to replace an owner if he does not wear his signer hat anymore as used in `_swapSigner`:

```
for (uint256 i; i < _ownerCount - 1;) {
    ownerToCheck = _owners[i];

    if (!isValidSigner(ownerToCheck)) {
        // prep the swap
        data = abi.encodeWithSignature(
            "swapOwner(address,address,address)",
            ...
```

## Impact

bypass restrictions and perform action that should be disallowed.

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L507-L529

## Tool used

Manual Review

## Recommendation

Perform a pre- and post-flight comparison on the safe owners, analogous to what is currently done with the modules.

## Discussion

**spengrah**

https://github.com/Hats-Protocol/hats-zodiac/pull/5

# Issue M-17: attacker can perform malicious transactions in the safe because reentrancy is not implemented correctly in the checkTransaction() and checkAfterExecution() function in HSG

Source: https://github.com/sherlock-audit/2023-02-hats-judging/issues/124

## Found by

unforgiven, cducrest-brainbot

## Summary

to prevent reentrancy during the safe's `execTransaction()` function call code use `_guardEntries` and increase it in the `checkTransaction()` and decrease it in the `checkAfterExecution()`. but the logic is wrong and code won't underflow in the `checkAfterExecution()` if attacker perform reentrancy during the `execTransaction()`

## Vulnerability Detail

This is some part of the `checkTransaction()` and `checkAfterExecution()` code:

```
function checkTransaction(
    address to,
    uint256 value,
    bytes calldata data,
    Enum.Operation operation,
    uint256 safeTxGas,
    uint256 baseGas,
    uint256 gasPrice,
    address gasToken,
    address payable refundReceiver,
    bytes memory signatures,
    address // msgSender
) external override {
    if (msg.sender != address(safe)) revert NotCalledFromSafe();

    uint256 safeOwnerCount = safe.getOwners().length;
    // uint256 validSignerCount = _countValidSigners(safe.getOwners());

    // ensure that safe threshold is correct
    reconcileSignerCount();

    if (safeOwnerCount < minThreshold) {
        revert BelowMinThreshold(minThreshold, safeOwnerCount);
```

```
    }

    // get the tx hash; view function
    bytes32 txHash = safe.getTransactionHash(
        // Transaction info
        to,
        value,
        data,
        operation,
        safeTxGas,
        // Payment info
        baseGas,
        gasPrice,
        gasToken,
        refundReceiver,
        // Signature info
        // We subtract 1 since nonce was just incremented in the parent function
↪   call
        safe.nonce() - 1 // view function
    );

    uint256 validSigCount = countValidSignatures(txHash, signatures,
↪   signatures.length / 65);

    unchecked {
        ++_guardEntries;
    }
}

/// @notice Post-flight check to prevent `safe` signers from removing this
↪   contract guard, changing any modules, or changing the threshold
/// @dev Modified from https://github.com/gnosis/zodiac-guard-mod/blob/988ebc7b7
↪   1e352f121a0be5f6ae37e79e47a4541/contracts/ModGuard.sol#L86
function checkAfterExecution(bytes32, bool) external override {
    if (msg.sender != address(safe)) revert NotCalledFromSafe();

    // leave checked to catch underflows triggered by re-erntry attempts
    --_guardEntries;
}
```

as you can see code increase the value of the `_guardEntries` in the `checkTransaction()` which is called before the transaction execution and decrease its value in the `checkAfterExecution` which is called after transaction execution. this won't protect against reentrancy during the safe's `execTransaction()` call. attacker can perform this actions:

1. Transaction1 which has valid number of signers and set the value of the guard



to 0x0. and call `safe.execTransaction(Transaction2)`.

2. Transaction2 which reset the value of the guard to the HSG address.
3. now by calling `Tsafe.execTransaction(Transaction1)` code would first call `checkTransaction()` and would see the number of the signers is correct and then increase the value of the `_guardEntiries` to 1 and then code in safe would execute the Transaction1 which would set the guard to 0x0 and execute the Transaction2 in safe.
4. because guard is 0x0 code would execute the Transaction2 and then during that code would re-set the value of the guard to the HSG address.
5. now `checkAfterExecution()` would get exeucted and would see that guard value is correct and would decrease the `_guardEntiries`

the attack is possible by changing the value of the `threshhold` in the safe. because code would perform two increase and one decrease during the reentrancy so the underflow won't happen.

## Impact

it's possible to set guard or threshold during the execTransaction() and execute another malicious transaction which resets guard and threshold

## Code Snippet

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L507-L540

https://github.com/Hats-Protocol/hats-zodiac/blob/9455cc0957762f5dbbd8e62063d970199109b977/src/HatsSignerGateBase.sol#L500-L503

https://github.com/safe-global/safe-contracts/blob/cb22537c89ea4187f4ad141ab2e1abf15b27416b/contracts/Safe.sol#L172-L174

## Tool used

Manual Review

## Recommendation

set the value of the guard to 1 and decrease in the `checkTransaction()` and increase in the `checkAfterExecution()`.