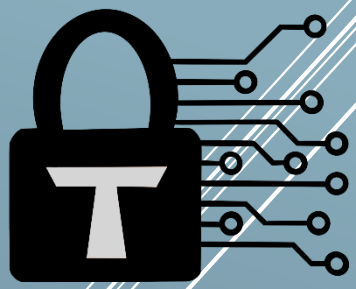
Trust Security

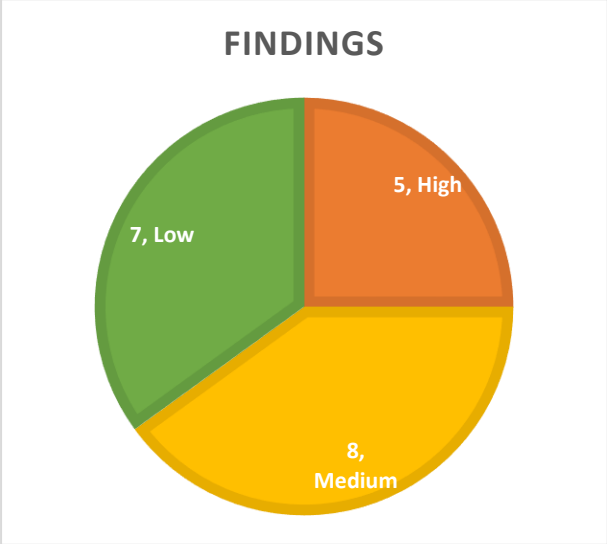


Smart Contract Audit

Hats Protocol

20/02/23

Executive summary

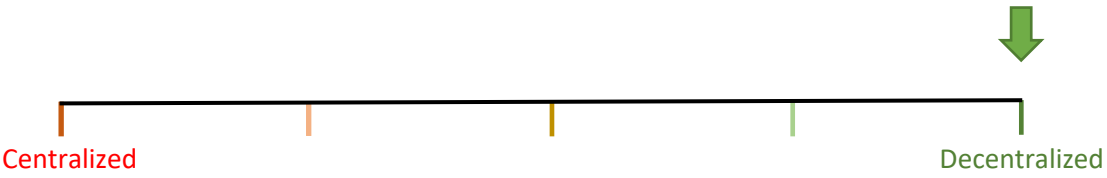


Category	DAO management
Audited file count	7
Lines of Code	1231
Auditor	Trust
Time period	12/02-20/02

Findings

Severity	Total	Fixed	Fix issues	Acknowledged	Disputed
High	5	5	1	-	-
Medium	8	6	1	2	-
Low	7	6	1	1	-

Centralization score



Signature

Digitally signed by Trust
Date: 2023-02-23 18:07:38

Document properties

Versioning

Version	Date	Description
0.1	20/02/23	Client report
0.2	23/02/23	Team response + mitigation review

Contact

Or Cyngiser, AKA Trust

boss@trustindistrust.com

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

hats-zodiac

- src/HatsSignerGate.sol
- src/HatsSignerGateBase.sol
- src/HatsSignerGateFactory.sol
- src/HSGLib.sol
- src/MultiHatsSignerGate.sol

hats-protocol

- src/Hats.sol
- src/HatsIdUtilities.sol

Repository details

hats-zodiac

- **Repository URL:** https://github.com/Hats-Protocol/hats-zodiac
- **Commit hash:** d4b878bd4bd8f8c0c735bbced127d18b63ade43e
- **Fix commit hash:** b9b7fcf22fd5cbb98c7d93dead590e80bf9c780a

hats-protocol

- **Repository URL:** https://github.com/Hats-Protocol/hats-protocol
- **Commit hash:** 60f07df0679ba52d4ad818b1bb3700d2f4f5a63a
- **Fix commit hash:** 37fa2d2f3468ca6959da8e1bc17e7fcc3bb07961

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Project reduced complexity, lowering overall attack risk.
Documentation	**Excellent**	Project specs and code are well documented.
Best practices	**Excellent**	Project adheres to latest industry standards. Code is structured very well.
Centralization risks	**Excellent**	Project is highly permissionless and allows clients to optimize for the desired amount of centralization.

Findings

High severity findings

TRST-H-1 More than one hat of the same hatId can be assigned to a user

- **Category:** Logical flaw
- **Source:** Hats.sol
- **Status:** Fixed

Description

Hats are minted internally using *_mintHat()*.

```
/// @notice Internal call to mint a Hat token to a wearer
/// @dev Unsafe if called when `_wearer` has a non-zero balance of
`_hatId`
/// @param _wearer The wearer of the Hat and the recipient of the
newly minted token
/// @param _hatId The id of the Hat to mint
function _mintHat(address _wearer, uint256 _hatId) internal {
    unchecked {
        // should not overflow since `mintHat` enforces max balance
of 1
        _balanceOf[_wearer][_hatId] = 1;
        // increment Hat supply counter
        // should not overflow given AllHatsWorn check in `mintHat`
        ++_hats[_hatId].supply;
    }
    emit TransferSingle(msg.sender, address(0), _wearer, _hatId, 1);
}
```

As documentation states, it is unsafe if **_wearer** already has the **hatId**. However, this could easily be the case when called from *mintHat()*.

```
function mintHat(uint256 _hatId, address _wearer) public returns
(bool) {
    Hat memory hat = _hats[_hatId];
    if (hat.maxSupply == 0) revert HatDoesNotExist(_hatId);
    // only the wearer of a hat's admin Hat can mint it
    _checkAdmin(_hatId);
    if (hat.supply >= hat.maxSupply) {
        revert AllHatsWorn(_hatId);
    }
    if (isWearerOfHat(_wearer, _hatId)) {
        revert AlreadyWearingHat(_wearer, _hatId);
    }
    _mintHat(_wearer, _hatId);
    return true;
}
```

The function validates **_wearer** doesn't currently wear the hat, but its balance could still be over 0, if the hat is currently toggled off or the wearer is not eligible.

The impact is that the hat supply is forever spent, while nobody actually received the hat. This could be used maliciously or occur by accident. When the hat is immutable, the max supply can never be corrected for this leak. It could be used to guarantee no additional, unfriendly hats can be minted to maintain permanent power.

Recommended mitigation

Instead of checking if user currently wears the hat, check if its balance is over 0.

Team response

Accepted.

Mitigation review

Fixed by checking the static hat balance of wearer.

TRST-H-2 TXs can be executed by less than the minimum required signatures

- **Category:** Logical flaws
- **Source:** HatsSignerGateBase.sol
- **Status:** Fixed

Description

In HatsSignerGateBase, *checkTransaction()* is the function called by the Gnosis safe to approve the transaction. Several checks are in place.

```
uint256 safeOwnerCount = safe.getOwners().length;
if (safeOwnerCount < minThreshold) {
    revert BelowMinThreshold(minThreshold, safeOwnerCount);
}
```

```
uint256 validSigCount = countValidSignatures(txHash, signatures,
signatures.length / 65);
// revert if there aren't enough valid signatures
if (validSigCount < safe.getThreshold()) {
    revert InvalidSigners();
}
```

The first check is that the number of owners registered on the safe is at least **minThreshold**. The second check is that the number of valid signatures (wearers of relevant hats) is not below the safe's threshold. However, it turns out these requirements are not sufficient. A possible situation is that there are plenty of owners registered, but currently most do not wear a hat. reconcileSignerCount() could be called to reduce the safe's threshold to the current validSigCount, which can be below **minThreshold**. That would make both the first and second check succeed. However, **minThreshold** is defined to be the smallest number of signers that must come together to make a TX. The result is that a single signer could execute a TX on the safe, if the other signers are not wearers of hats (for example, their toggle has been temporarily set off in the case of multi-hat signer gate).

Recommended mitigation

Add another check in *checkTransaction*(), which states that **validSigCount >= minThreshold**.

Team response

Accepted.

Mitigation review

Fixed.

TRST-H-3 Target signature threshold can be bypassed leading to minority TXs

- **Category:** Logical flaws
- **Source:** HatsSignerGateBase.sol
- **Status:** Fixed

Description

checkTransaction() is the enforcer of the HSG logic, making sure signers are wearers of hats and so on. The check below makes sure sufficient hat wearers signed the TX:

```
uint256 validSigCount = countValidSignatures(txHash, signatures,
signatures.length / 65);
// revert if there aren't enough valid signatures
if (validSigCount < safe.getThreshold()) {
    revert InvalidSigners();
}
```

The issue is that the safe's threshold is not guaranteed to be up to date. For example, initially there were 5 delegated signers. At some point, three lost eligibility. *reconcileSignerCount*() is called to update the safe's threshold to now have 2 signers. At a later point, the three signers which lost eligibility regained it. At this point, the threshold is still two, but there are 5 valid signers, so if **targetThreshold** is not below 5, they should all sign for a TX to be executed. That is not the case, as the old threshold is used. There are various scenarios which surface the lack of synchronization between the wearer status and safe's stored threshold.

Recommended mitigation

Call *reconcileSignerCount()* before the validation code in *checkTransaction()*.

Team response

Fixed.

TRST-H-4 maxSigners can be bypassed

- **Category:** Logical flaws
- **Source:** HatsSignerGate.sol / MultiHatsSignerGate.sol
- **Status:** Fixed + new issue

Description

maxSigners is specified when creating an HSG and is left constant. It is enforced in two ways – **targetThreshold** may never be set above it, and new signers cannot register to the HSG when the signer count reached **maxSigners**. Below is the implementation code in HatsSignerGate.

```
function claimSigner() public virtual {
    if (signerCount == maxSigners) {
        revert MaxSignersReached();
    }
    if (safe.isOwner(msg.sender)) {
        revert SignerAlreadyClaimed(msg.sender);
    }
    if (!isValidSigner(msg.sender)) {
        revert NotSignerHatWearer(msg.sender);
    }
    _grantSigner(msg.sender);
}
```

An issue that arises is that this doesn't actually limit the number of registered signers. Indeed, **signerCount** is a variable that can fluctuate when wearers lose eligibility or a hat is inactive. At this point, *reconcileSignerCount()* can be called to update the signerCount to the current valid wearer count. A simple attack which achieves unlimited claims is as follows:

1. Assume **maxSigners** = 10
2. 10 signers claim their spot, so **signerCount** is maxed out
3. A signer misbehaves, loses eligibility and the hat.
4. *reconcile*() is called, so **signerCount** is updated to 9
5. A new signer claims, making **signerCount** = 10
6. The malicious signer behaves nicely and regains the hat.
7. *reconcile*() is called again, making **signerCount** = **11**
8. At this point, any eligible hat wearer can claim their hat, easily overrunning the **maxSigners** restriction.

Recommended mitigation

The root cause is that users which registered but lose their hat are still stored in the safe's owners array, meaning they can always get re-introduced and bump the **signerCount**. Instead of checking the **signerCount**, a better idea would be to compare with the list of owners saved on the safe. If there are owners that are no longer holders, *removeSigner()* can be called to vacate space for new signers.

Team response

Accepted; added a *swapSigner()* flow to *claimSigner().*

Mitigation review

Fixed but introduced a new issue. The new code will swap the new signer with an invalid old signer.

```
address[] memory owners = safe.getOwners();
uint256 ownerCount = owners.length;
if (ownerCount >= maxSigs) {
    _swapSigner(owners, ownerCount, maxSigs, currentSignerCount,
msg.sender);
} else {
    _grantSigner(owners, currentSignerCount, msg.sender);
}
```

However, it's possible that all current owners are valid signers, in this case *_swapSigner()* will complete the loop and return gracefully. A user will think they have claimed signer successfully, but nothing has changed.

TRST-H-5 Minority may be able to call safe operations

- **Category:** Logical flaws
- **Source:** HatsSignerGateBase.sol
- **Status:** Fixed

Description

Users can update the HSG's view of signers using *reconcileSignerCount()*

```
function reconcileSignerCount() public {
    address[] memory owners = safe.getOwners();
    uint256 validSignerCount = _countValidSigners(owners);
    // update the signer count accordingly
    signerCount = validSignerCount;
    if (validSignerCount <= targetThreshold && validSignerCount !=
safe.getThreshold()) {
        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();
        }
    }
}
```

Notice that the safe's registered threshold is only updated if the new **validSignerCount** is lower than the **targetThreshold**. Actually, that is not desired behavior, because if signers have reactivated or have become eligible again, it's possible this condition doesn't hold, and the previous threshold could be *lower than targetThreshold*. In this scenario, a small minority could still execute TXs when **targetThreshold** signatures are needed.

Recommended mitigation

Add an else clause, stating that if the new **validSignerCount > targetThreshold** and **safe.getThreshold() < targetThreshold**, the threshold changes to **targetThreshold**.

Team response

Accepted.

Mitigation review

Fixed by restructuring conditions in *reconcileSignerCount()*.

Medium severity findings

TRST-M-1 Hats token breaks ERC1155 specifications

- **Category:** Specs compatibility issues
- **Source:** Hats.sol
- **Status:** Acknowledged

Description

The Hats token implements ERC1155. It implements *safeTransferFrom()* and *batchSafeTransferFrom()* as revert-only functions, so tokens cannot be transferred using standard ERC1155 means. However, hats can still be transferred using *mintHat(), mintTopHat()* and *transferHat()*. Whenever there is a transfer, the standard requires checking the receiver accepts the transfer:

> "If an implementation specific API function is used to transfer ERC-1155 token(s) to a contract, the `safeTransferFrom` or `safeBatchTransferFrom` (as appropriate) rules MUST still be followed if the receiver implements the `ERC1155TokenReceiver` interface. If it does not the non-standard implementation SHOULD revert but MAY proceed."

By not checking a contract receiver accepts the transfer, Hats token does not adhere to ERC1155.

Recommended mitigation

If the recipient implements ERC1155TokenReceiver, require that it accepts the transfer. If the recipient is a contract that does not implement a receiver, reject the operation.

Team response

Acknowledged; changed documentation to ERC1155-similar and to explicitly clarify that Hats implements the ERC1155 interface but does not conform to the full standard.

TRST-M-2 Attacker can DOS minting of new top hats in low-fee chains

- **Category:** Insufficient DOS protection
- **Source:** Hats.sol
- **Status:** Acknowledged

Description

In Hats protocol, anyone can be assigned a top hat via the *mintTopHat()* function. The top hats are structured with top 32 bits acting as a domain ID, and the lower 224 bits are cleared. There are therefore up to 2^32 = ~ 4 billion top hats. Once they are all consumed, mintTopHat() will always fail:

```
// uint32 lastTopHatId will overflow in brackets
topHatId = uint256(++lastTopHatId) << 224;
```

This behavior exposes the project to a DOS vector, where an attacker can mint 4 billion top hats in a loop and make the function unusable, forcing a redeploy of Hats protocol. This is unrealistic on ETH mainnet due to gas consumption, but definitely achievable on the cheaper L2 networks. As the project will be deployed on a large variety of EVM blockchains, this poses a significant risk.

Recommended mitigation

Require a non-refundable deposit fee (paid in native token) when minting a top hat. Price it so that consuming the 32-bit space will be impossible. This can also serve as a revenue stream for the Hats project.

Team response

Acknowledged; electing not to address in v1 for several reasons:

1. Additional requirement to set & manage authorization for withdrawal
2. Challenge of setting a consistently reasonable fee on chains without stablecoin-based native tokens (i.e. all except for Gnosis Chain) / the added complexity and centralization risk of making the fee adjustable
3. Contract size constraints

TRST-M-3 Linking of hat trees can freeze hat operations

- **Category:** overflow issues
- **Source:** Hats.sol
- **Status:** Fixed

Description

Hats support tree-linking, where hats from one node link to the first level of a different domain. This way, the amount of levels for the linked-to tree increases by the linked-from level count. This is generally fine, however lack of checking of the new total level introduces severe risks.

```
/// @notice Identifies the level a given hat in its hat tree
/// @param _hatId the id of the hat in question
/// @return level (0 to type(uint8).max)
function getHatLevel(uint256 _hatId) public view returns (uint8) {
```

The *getHatLevel()* function can only return up to level 255. It is used by the *checkAdmin()* call used in many of the critical functions in the Hats contract. Therefore, if for example, 17 hat

domains are joined together in the most stretched way possible, It would result in a correct hat level of 271, making this calculation revert:

```
if (treeAdmin != 0) {
    return 1 + uint8(i) + getHatLevel(treeAdmin);
}
```

The impact is that intentional or accidental linking that creates too many levels would freeze the higher hat levels from any interaction with the contract.

Recommended mitigation

It is recommended to add a check in *_linkTopHatToTree(),* that the new accumulated level can fit in uint8. Another option would be to change the maximum level type to uint32.

Team response

Accepted; increased max level type to uint32.

Mitigation review

Fixed.

TRST-M-4 Admin can transfer hat to a non-eligible target, potentially burning the hat

- **Category:** Logical flaws
- **Source:** Hats.sol
- **Status:** Fixed

Description

Hat admins may transfer child hats using *transferHat()*. It checks the hat receiver does not currently have a balance for this **hatId**.

```
// Check if recipient is already wearing hat; also checks storage to
maintain balance == 1 invariant
if (_balanceOf[_to][_hatId] > 0) {
    revert AlreadyWearingHat(_to, _hatId);
}
```

The issue is that it does not also check that the recipient is eligible for the **hatId**. Therefore, an admin could transfer a hat and then it could be immediately burnt by anyone using the *checkHatWearerStatus()* call.

Recommended mitigation

Verify the recipient is eligible for the **hatId** before transferring it.

Team response

Accepted.

Mitigation review

Fixed.

TRST-M-5 Attacker can make a signer gate creation fail

- **Category:** Frontrunning issues
- **Source:** HatsSignerGateFactory.sol
- **Status:** Fixed

Description

DAOs can deploy a HSG using *deployHatsSignerGateAndSafe()* or *deployMultiHatsSignerGateAndSafe()*.The parameters are encoded and passed to *moduleProxyFactory.deployModule()*:

```
bytes memory initializeParams = abi.encode(
    _ownerHatId, _signersHatId, _safe, hatsAddress, _minThreshold,
_targetThreshold, _maxSigners, version
);
hsg = moduleProxyFactory.deployModule(
    hatsSignerGateSingleton, abi.encodeWithSignature("setUp(bytes)",
initializeParams), _saltNonce
);
```

This function will call *createProxy()*:

```
proxy = createProxy(
    masterCopy,
    keccak256(abi.encodePacked(keccak256(initializer), saltNonce))
);
```

The second parameter is the generated salt, which is created from the **initializer** and passed **saltNonce**. Finally *createProxy*() will use CREATE2 to create the contract:

```
function createProxy(address target, bytes32 salt)
    internal
    returns (address result)
{
    if (address(target) == address(0)) revert ZeroAddress(target);
    if (address(target).code.length == 0) revert
TargetHasNoCode(target);
    bytes memory deployment = abi.encodePacked(
        hex"602d8060093d393df3363d3d373d3d3d363d73",
        target,
        hex"5af43d82803e903d91602b57fd5bf3"
    );
    // solhint-disable-next-line no-inline-assembly
    assembly {
        result := create2(0, add(deployment, 0x20),
mload(deployment), salt)
    }
    if (result == address(0)) revert TakenAddress(result);
}
```

An issue could be that an attacker can frontrun the creation TX with their own creation request, with the same parameters. This would create the exact address created by the CREATE2 call, since the parameters and therefore the final salt will be the same. When the victim's transaction would be executed, the address is non-empty so the EVM would reject its creation. This would result in a bad UX for a user, who thinks the creation did not

succeed. The result contract would still be usable, but would be hard to track as it was created in another TX.

Recommended mitigation

Use an ever-increasing nonce counter to guarantee unique contract addresses.

Team response

Accepted.

Team response

Fixed.

TRST-M-6 Signers can backdoor the safe to execute any transaction in the future without consensus

- **Category:** Lack of safety checks
- **Source:** HatsSignerGateBase.sol
- **Status:** Partial fix

Description

The function *checkAfterExecution()* is called by the safe after signer's request TX was executed (and authorized). It mainly checks that the linkage between the safe and the HSG has not been compromised.

```
function checkAfterExecution(bytes32, bool) external override {
    if (

abi.decode(StorageAccessible(address(safe)).getStorageAt(uint256(GUAR
D_STORAGE_SLOT), 1), (address))
            != address(this)
    ) {
        revert CannotDisableThisGuard(address(this));
    }
    if (!IAvatar(address(safe)).isModuleEnabled(address(this))) {
        revert CannotDisableProtectedModules(address(this));
    }
    if (safe.getThreshold() != _correctThreshold()) {
        revert SignersCannotChangeThreshold();
    }
    // leave checked to catch underflows triggered by re-erntry
attempts
    --guardEntries;
}
```

However, it is missing a check that no new modules have been introduced to the safe. When modules execute TXs on a Gnosis safe, the guard safety callbacks do not get called. As a result, any new module introduced is free to execute whatever it wishes on the safe. It constitutes a serious backdoor threat and undermines the HSG security model.

Recommended mitigation

Check that no new modules have been introduced to the safe, using the *getModulesPaginated()* utility.

Team response

Accepted; added a method for HSG owner to add modules, and an enabled modules counter to check against in *checkAfterTransaction*()

Mitigation review

Fix is not bulletproof. A malicious transaction can remove an existing module and replace it with their own malicious module. In addition to a length check on the modules array, it is necessary to do a full comparison before and after the TX execution.

TRST-M-7 Hats can't be renounced when not worn, leading to abuse concerns

- **Category:** Logical flaws
- **Source:** Hats.sol
- **Status:** Fixed

Description

Hats can be renounced by the owner using *renounceHat()* call. They can only be renounced when currently worn, regardless if they have a positive balance. Issues can arise from abuse by **toggle** or **eligibility** delegates. They can temporarily disable or sanction the wearer so that it cannot be renounced. At a later point, when the wearer is to be made accountable for their responsibilities, they could be toggled back on and penalize an innocent hat wearer.

Recommended mitigation

Allow hats to be renounced even when they are not worn right now. A different event parameter can be used to display if they were renounced while worn or not.

Team response

Accepted.

Mitigation review

Fixed by applying the suggested mitigation.

TRST-M-8 Safety checks compare safe's threshold with a stale value

- **Category:** Insufficient safety checks
- **Source:** HatsSignerGateBase.sol
- **Status:** Fixed

Description

In HatsSignerGateBase, *_correctThreshold()* calculates what the safe's threshold should be. However, it uses **signerCount** without updating it by calling *reconcileSignerCount()*. Therefore, in *checkAfterExecution()*, the safe's current threshold will be compared to

potentially the wrong value. This may trip valid transactions or allow malicious ones to go through, where the threshold should end up being higher.

Recommended mitigation

Call *reconcileSignerCount()* before making use of the **signerCount** value.

Team response

Accepted; added *_countValidSigners()* rather than *reconcileSignerCount().*

Mitigation review

Fixed.

Low severity findings

TRST-L-1 createHat does not detect MAX_LEVEL admin correctly

- **Category:** Integer trimming flaws
- **Source:** Hats.sol
- **Status:** Fixed

Description

In *createHat()*, the contract checks user is not minting hats for the lowest hat tier:

```
function createHat(
    uint256 _admin,
    string memory _details,
    uint32 _maxSupply,
    address _eligibility,
    address _toggle,
    bool _mutable,
    string memory _imageURI
) public returns (uint256 newHatId) {
    if (uint8(_admin) > 0) {
        revert MaxLevelsReached();
    }
    ….
}
```

The issue is that it does not check for max level correctly, as it looks only at the lowest 8 bits. Each level is composed of 16 bits, so ID xx00 would pass this check.

Fortunately, although the check is passed, the function will revert later. The call to *getNextId(_admin)* will return **0** for max-level admin, and *_checkAdmin(0)* is guaranteed to fail. However, the check should still be fixed as it is not exploitable only by chance.

Recommended mitigation

Change the conversion to uint16.

Team response

Accepted.

Mitigation review

Fixed.

TRST-L-2 Incorrect imageURI is returned for hats in certain cases

- **Category:** Off-by-one flaws
- **Source:** Hats.sol
- **Status:** Fixed

Description

Function *getImageURIForHat()* should return the most relevant **imageURI** for the requested hatId. It will iterate backwards from the current level down to level 0, and return an image if it exists for that level.

```
function getImageURIForHat(uint256 _hatId) public view returns
(string memory) {
    // check _hatId first to potentially avoid the `getHatLevel` call
    Hat memory hat = _hats[_hatId];
    string memory imageURI = hat.imageURI; // save 1 SLOAD
    // if _hatId has an imageURI, we return it
    if (bytes(imageURI).length > 0) {
        return imageURI;
    }
    // otherwise, we check its branch of admins
    uint256 level = getHatLevel(_hatId);
    // but first we check if _hatId is a tophat, in which case we
fall back to the global image uri
    if (level == 0) return baseImageURI;
    // otherwise, we check each of its admins for a valid imageURI
    uint256 id;
    // already checked at `level` above, so we start the loop at
`level - 1`
    for (uint256 i = level - 1; i > 0;) {
        id = getAdminAtLevel(_hatId, uint8(i));
        hat = _hats[id];
        imageURI = hat.imageURI;
        if (bytes(imageURI).length > 0) {
            return imageURI;
        }
        // should not underflow given stopping condition is > 0
        unchecked {
            --i;
        }
    }
    // if none of _hatId's admins has an imageURI of its own, we
again fall back to the global image uri
    return baseImageURI;
}
```

It can be observed that the loop body will not run for level 0. When the loop is finished, the code just returns the **baseImageURI**, which is a Hats-level fallback, rather than top hat level

fallback. As a result, the image displayed will not be correct when querying for a level above 0, when all levels except level 0 have no registered image.

Recommended mitigation

Before returning the **baseImageURI**, check if level 0 admin has a registered image.

Team response

Accepted.

Team response

Fixed using an additional check.

TRST-L-3 Fetching of hat status may fail due to lack of input sanitization

- **Category:** Input sanitization flaw
- **Source:** Hats.sol
- **Status:** Fixed

Description

The functions *_isActive()* and *_isEligible()* are used by *balanceOf()* and other functions, so they should not ever revert. However, they perform ABI decoding from external inputs.

```
function _isActive(Hat memory _hat, uint256 _hatId) internal view
returns (bool) {
    bytes memory data =
abi.encodeWithSignature("getHatStatus(uint256)", _hatId);
    (bool success, bytes memory returndata) =
_hat.toggle.staticcall(data);
    if (success && returndata.length > 0) {
        return abi.decode(returndata, (bool));
    } else {
        return _getHatStatus(_hat);
    }
}
```

If **toggle** returns invalid return data (whether malicious or by accident), *abi.decode()* would revert causing the entire function to revert.

Recommended mitigation

Wrap the decoding operation for both affected functions in a try/catch statement. Fall back to the *_getHatStatus()* result if necessary. Checking that **returndata** size is correct is not enough as bool encoding must be 64-bit encoded 0 or 1.

Team response

Accepted.

Mitigation review

Fixed by performing safe decoding of input data, and falling back to the static hat status or standing.

TRST-L-4 Admin check is overly gas-intensive

- **Category:** Gas intensive operations
- **Source:** Hats.sol
- **Status:** Fixed + new issue

Description

Hats checks for most operations that the caller is an authorized hat admin. The function implementing this check is *isAdminOfHat()*. The function loops and checks if the sender is a wearer of a lower-level hat.

```
while (adminHatLevel > 0) {
    if (isWearerOfHat(_user, getAdminAtLevel(_hatId, adminHatLevel)))
{
        return true;
    }
    // should not underflow given stopping condition > 0
    unchecked {
        --adminHatLevel;
    }
}
```

The issue is that calling *getAdminAtLevel()* for every level is very gas intensive. The vast majority of the cost of the function is resolving the hat level using *getHatLevel()*, but it is implemented recursively. Most of the gas can be saved by refactoring this code, which will be used by almost every interaction with Hats.

Recommended mitigation

Refactor some part of the described code flow so that it will be less gas intensive for hats with linked trees.

Team response

Accepted; refactored to increase efficiency

Mitigation review

Fixed, but introduced new issue. The new implementation checks at the end of *isAdminOfHat*() if the hat is linked to another tree.

```
// if we get here, we're at the top of _hatId's local tree
linkedTreeAdmin = linkedTreeAdmins[getTophatDomain(_hatId)];
if (linkedTreeAdmin == 0) {
    // tree is not linked
    return isWearerOfHat(_user, getLocalAdminAtLevel(_hatId, 0));
} else {
    if (isWearerOfHat(_user, linkedTreeAdmin)) return true; // user
wears the linkedTreeAdmin
    else return isAdminOfHat(_user, linkedTreeAdmin); // check if
user is admin of linkedTreeAdmin (recursion)
}
```

Importantly, it is never checked if user is wearer of the admin hat at level 0, in the event that the tree is linked. The impact is lack of adminship at the top of the tree.

TRST-L-5 Lack of zero-address check for important parameters

- **Category:** Input sanitization flaws
- **Source:** Hats.sol
- **Status:** Fixed

Description

Hats allows admin hats to change the **toggle** and **eligibility** set for a given hat. For example:

```
function changeHatToggle(uint256 _hatId, address _newToggle) external
{
    _checkAdmin(_hatId);
    Hat storage hat = _hats[_hatId];
    if (!_isMutable(hat)) {
        revert Immutable();
    }
    hat.toggle = _newToggle;
    emit HatToggleChanged(_hatId, _newToggle);
}
```

The code lacks a zero-address check for **_newToggle**. There is no good reason why this value might need to be set to zero, and it's a standard best practice for fault-detection.

Recommended mitigation

Add a zero-address check for both address parameters.

Team response

Accepted

Mitigation review

Fixed.

TRST-L-6 Safe's registered threshold could be below minThreshold

- **Category:** Lack of safety precautions
- **Source:** HatsSignerGateBase.sol
- **Status:** Acknowledged

Description

The functions *setTargetThreshold()* and *reconcileSignerCount*() change the safe's registered threshold. Both functions do not check the new value is above or equal to **minThreshold**. This is not a serious issue if the HSG is defined to be the enforcer of the **minThreshold**. However, this was not done correctly as was described in H-2, so it would be best to never set the safe's version of the threshold to below **minThreshold**.

Recommended mitigation

Recommended mitigation steps

Team response

Acknowledged; not making explicit changes because the Safe contract does not allow thresholds below number of owners, so we can't remove all scenarios where threshold is below **minThreshold**. Risks associated with this finding should be mitigated by changes done applied in other findings.

TRST-L-7 Reentrancy guards can be easily bypassed

- **Category:** Lack of safety precautions
- **Source:** HatsSignerGateBase.sol
- **Status:** Fixed

Description

In *checkTransaction*(), **guardEntries** is incremented while in *checkAfterExecution*(), it is decremented, implementing a basic reentrancy guard.

The issue is that both functions lack a msg.sender check. Therefore, if there was a way to exploit the contract using a reentrancy, this defense would be futile. Attacker could simply call *checkTransaction()* an unlimited amount of times, with valid arguments. Then, *checkAfterExecution()* would never overflow from the reentrancy abuse.

Recommended mitigation

Check that the msg.sender is the safe for the two functions above.

Team response

Accepted.

Team response

Fixed.

Additional recommendations

Documentation issues

- Remove irrelevant comment in *buildHatId():*

```
/// @dev Check hats[_admin].lastHatId for the previous hat
created underneath _admin
```

- Correct comment in *checkHatStatus():*

```
// then we know the contract exists and has the getWearerStatus
function
```

- Remove comment in MultiHatsSignerGate's *claimSigner():*

```
/// @dev overloads HatsSignerGateBase.claimSigner()
```

- Change comment in *approveLinkTopHatToTree()* – can be approved by **wearer or** admin

```
/// @dev Requests can only be approved by an admin of the
`_newAdminHat`, and there
```

Cleaner code

- Consider using the utility function *getTophatDomain()* here:

```
uint256 treeAdmin = linkedTreeAdmins[uint32(_hatId >> (256 -
TOPHAT_ADDRESS_SPACE))];
```

- **SAFE_TX_TYPEHASH** is not used throughout the code, yet it is declared. Consider omitting it.
- Several variables which cannot be changed in the lifetime of the contract are not declared immutable (e.g. **maxSigners**). It wastes gas as well as potentially hurting the clarity of the code.

Emit events without state change

In *changeHatMaxSupply()*, if the new **maxSupply** equals the previous **maxSupply**, the function doesn't actually change anything yet emits a supply changed event. It is recommended to validate the next supply is greater than the current one.

Improving display of information

In *_constructURI(),* the printed JSON string is composed of some properties.

```
// split into two objects to avoid stack too deep error
string memory idProperties = string.concat(
    '"domain": "',
```

```
    LibString.toString(getTophatDomain(_hatId)),
    '", "id": "',
    LibString.toString(_hatId),
    '", "pretty id": "',
    "{id}",
    '",'
);
```

It is recommended to change the {id} placeholder to a more informative value, such as *LibString.toHexString(_hatId, 32)*.