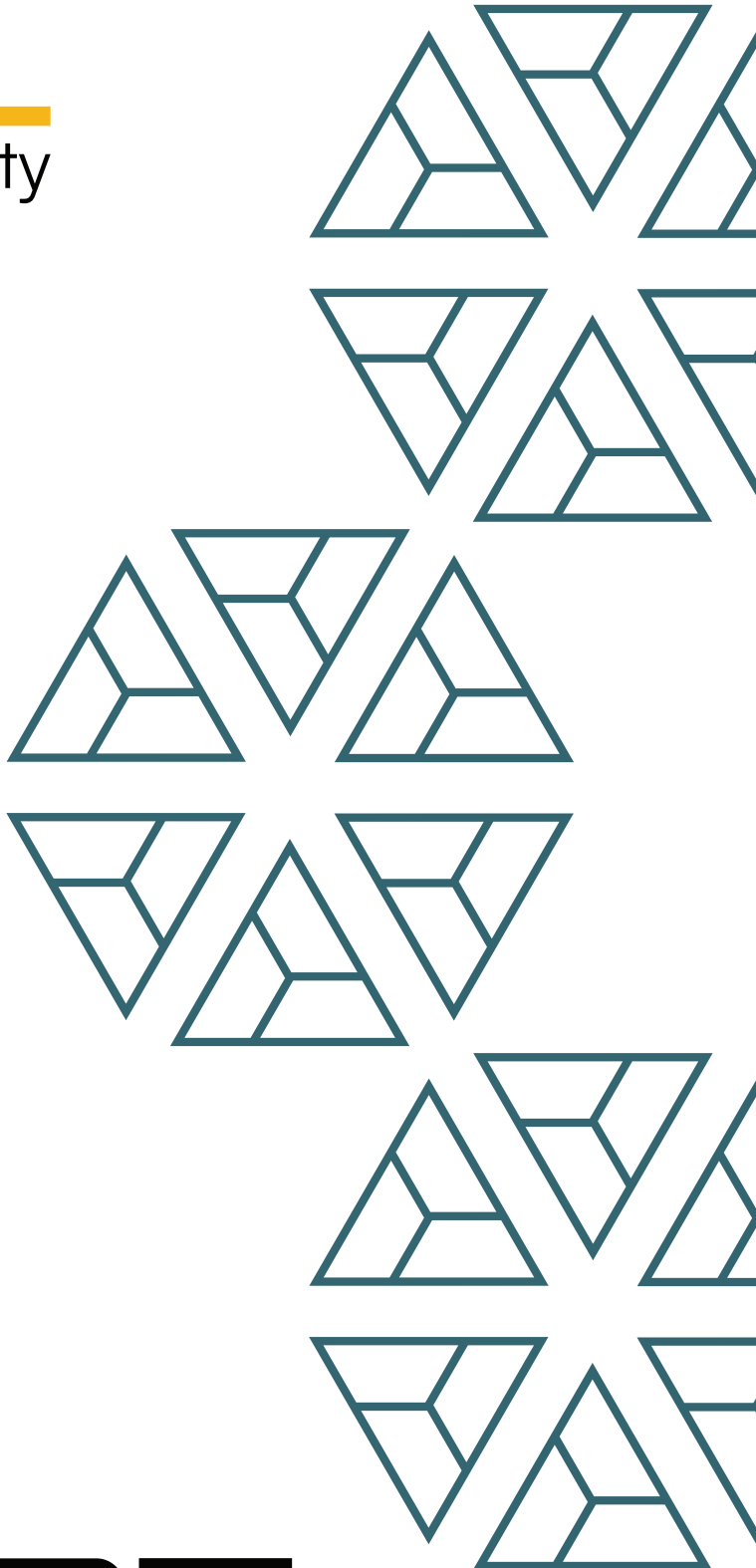
BAIL security



0x SafeGuard

FINAL REPORT

March '2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	0x - SafeGuard
Website	0x.org
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/0xProject/0x-settler/blob/0d5a970a9ba5dc02f1ec64793ac3cf0705222b4e/src/deployer/SafeGuard.sol
Resolution 1	https://github.com/0xProject/0x-settler/blob/3872df50064f997b4862b4b9b5770666688ef551/src/deployer/SafeGuard.sol

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution
High	3	2			
Medium	3			3	
Low	5			4	
Informational	6	1		4	
Governance					
Total	17	3		11	

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

SafeGuard

The SafeGuard contract is a transaction guard contract which can be integrated with the Safe contract. The standard Safe implementation includes a GuardManager which allows for adding a guard address.

This guard address is then consulted upon every execTransaction call, before and after the transaction is executed. More specifically, the checkTransaction function is invoked before the execution and the checkAfterExecution function is invoked after the execution.

The main goal of the SafeGuard contract is to prevent the execution of malicious transactions in the scenario where an attacker compromised the quorum of all keys to reach the threshold.

Usually, without the guard, a transaction is simply executed if the majority of owners have signed the transaction. With the guard in front, there are multiple safeguards, such as:

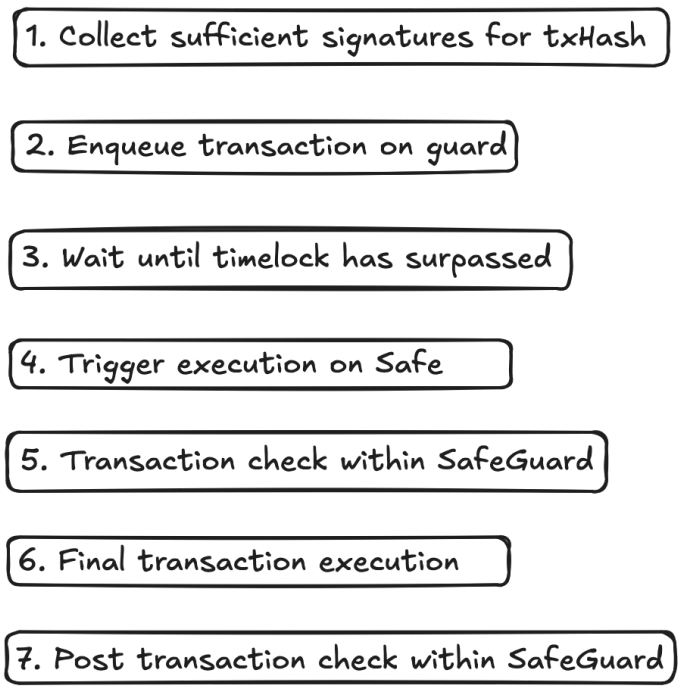
a) Disabling of delegatecalls
b) Timelock check
c) Module prevention

Once all these safeguards have been passed, a transaction can be executed. If for example a malicious transaction is queued, it can be prevented from being executed via the cancel function which simply invalidates the timelock for this txHash or via the lock function which is callable by any of the Safe owners and force locks the contract, which temporarily prevents any execution. The lock function is rather an emergency function which is ideally never triggered unless one or multiple keys are compromised.

The contract furthermore leverages the SafeLib library in an effort to properly encode a txHash as well as to fetch the owner count and the guard from the Safe's storage.

Appendix: Execution Flow

Below we will illustrate the execution flow of a transaction execution:



If any check is not satisfied, it will never be possible to trigger the final execution of a queued TX.

Appendix: Timelock

In an effort to prevent the immediate execution of transactions which ensures that a transaction is not malicious, a timelock mechanism has been implemented. It requires that a transaction which has reached the threshold to be queued via the enqueue function which will then add a delay to its execution. Once the delay has passed, the transaction can be executed.

In the scenario where a transaction is considered as malicious, already a single owner can invoke the cancel function which prevents the transaction from being executed, if the timelock has not yet passed yet.

The delay time is initially zero and can be set by the Safe contract.

Appendix: Lock Mechanism

The contract exposes an emergency mechanism in the scenario where one or more keys are compromised. Any owner of the Safe can invoke the lock function which then completely locks the SafeGuard. During a locked SafeGuard, it is not possible to execute a transaction due to the check within the checkAfterExecution function (even if the timelock has surpassed). It is furthermore not possible to enqueue TXs

It is possible to unlock the contract once all owners of the Safe have signed the corresponding TX. Only the threshold is insufficient to unlock the SafeGuard.

Locking requires the caller to sign the subsequent unlock TX with the current nonce. This will ensure no malicious signer can lock the contract and prevent subsequent unlocking.

Appendix: Data Encoding

In an effort to properly check the signature correctness, the transaction data must be converted into two variables:

a) txHashData
b) txHash

The way how this is done can be found in the Safe.execTransaction function:

1) The data, including SAFE_TX_TYPEHASH is encoded and hashed with keccak256 to receive the safeTxHash
2) The safeTxHash is encoded with bytes1(0x19), bytes1(0x01) and the domainSeparator which finally forms the txHashData
3) Then txHashData is hashed with keccak256 which forms the txHash

The txHash and txHashData is then used within checkSignatures

The exact same approach is used within the SafeLib library by the enqueue and checkTransaction functions.

Threat Model

Safe ⇔ SafeGuard

Once the SafeGuard has been setup as the Guard of the Safe, all executions, including changes to the Safe must go through the SafeGuard, which means, all txs must go through the Timelock period (except the call to SafeGuard.unlock(), this one can be executed immediately if all signers have signed the unlock hash)

Threat Model: Find a way to bypass this setup

Enqueuing txs

To enqueue a msgHash on the SafeGuard, at least `threshold` of signers must have signed the hash.

Threat Model: Find a way to enqueue the msgHash without having the threshold reached

Malicious TX

If an attacker is able to gain access to keys to reach the threshold, it is possible to enqueue a malicious TX and activate the timelock for the said. There are two distinct actions which can then be executed by any remaining non malicious signer to prevent the execution:

a) cancel the malicious TX
b) lockdown the contract

Threat Model: Find a way how the malicious party can still achieve the execution

Timelock

All transactions besides the unlock of the contract must go through the timelock

Threat Model: Find a way to bypass the timelock

Module

It is explicitly prevented to add modules to the safe as long as the guard remains in place

Threat Model: Find a way to add a module to the safe

Delegatecall

Delegatecalls are explicitly prohibited

Threat Model: Find a way to execute delegatecalls

Excessive Limitations

The contract implements multiple limitations, including but not limited to:

a) Unlock can only be called if all owners sign the TX
b) Every TX besides unlock must go through TL
c) No reentrancy call into the SafeGuard must happen

d) No TX can be executed if SafeGuard is locked down
e) The safe's masterCopy must always point to singleton
f) The SafeGuard must never be the owner of the safe
g) Every TX must be enqueued first
h) Enqueue only possible if threshold reached
i) Enqueue not possible if locked down

Threat Model: DoS the contract by exploiting a limitation against its desired use

Core Invariants:

INV 1: SafeGuard contract must be set as the guard of the Safe before the SafeFactory is called to deploy the SafeGuard

INV 2: SafeGuard must not be owner of the Safe

INV 3: SafeGuard must be deployed via _SAFE_SINGLETON_FACTORY

INV 4: The Safe must have not any modules

INV 5: setDelay, unlock, checkTransaction and checkAfterTransaction must only be called by the safe

INV 6: cancel and lockdown must only be callable by an owner of the safe with a valid signature for subsequent unlock

INV 7: cancel is only possible if timelockEnd has not passed

INV 8: enqueue can only be called if contract is not locked down and SafeGuard was not removed

INV 9: enqueue can only be called with sufficient valid signatures for txHash

INV 10: enqueue must revert if operation is no normal call

INV 11: The same txHash can only be enqueued once

INV 12: checkTransaction must revert if _reentrancyGuard = true

INV 13: checkTransaction must revert if operation is no normal call

INV 14: unlock call requires unanimous signatures

INV 15: checkTransaction must revert if timelockEnd has not passed

INV 16: checkAfterExecution must revert if _reentrancyGuard = false

INV 17: checkAfterExecution must revert if SafeGuard is locked down

INV 18: checkAfterExecution must revert if Safe's masterCopy does not match _SINGLETON

INV 19: checkAfterExecution must revert if Safe has any modules

INV 20: checkAfterExecution must revert if SafeGuard is set to an owner of the safe

INV 21: Once _guardRemoved is set to true, the SafeGuard contract should never be set again

INV 22: getPrevOwner must always return the previous owner

INV 23: _removeOwnerTxHash must always be called with previous and oldOwner

INV 24: cancel can only be called if resign tx has been signed

INV 25: checkTransaction must always pass if signatures are unanimous

INV 26: lockDown must always check for validity of subsequent removeOwner signature if caller has canceled a transaction before

Privileged Functions

- setDelay
- checkTransaction
- checkAfterExecution
- unlock

Issue_01	A single malicious signer can DoS the entire Safe by canceling enqueued transactions
Severity	**High**
Description	Using the cancel() function on the SafeGuard allows a single owner of the Safe to be able to cancel any enqueued tx on the SafeGuard. Since the moment the SafeGuard is installed on the Safe, all the tx that the Safe will execute must pass through the SafeGuard, this means, the same single malicious signer can continuously keep canceling all tx. This allows for (among other scenarios) two things: a) The signer can simply prevent himself from being kicked out b) The signer can trivially brick the whole Safe, making it impossible to execute any logic. Moreover, it is also possible for a single owner to lock a contract (which can then be unlocked) but again backrun by the malicious signer to again lock the contract - indefinitely. An attacker can even cancel all TXs if the contract is locked (which is usually a good measurement but in that case fires back).
Recommendations	Consider adding a case on the SafeGuard that could allow calls to the Safe (including removeOwner/swapOwner) if all Safe's signers minus 1 have signed the message. In case of a more strict validation, it could also be considered to allow that if all owners have signed (which should always be possible because the original owner most of the time still has the key - even if hijacked).
Comments / Resolution	Resolved, it is now enforced that the caller of the cancel function must sign the _removeOwnerTxHash. Following this approach, it is ensured that a malicious actor can only cancel a transaction once and in the subsequent execution it is now allowed to remove the malicious actor.

	If however at any point there are two malicious actors in the system and the second actor denies to sign the signature, this attack would still be possible. However, it is rather rare for such a circumstance to be present.

Issue_02	Lack of possibility to execute emergency transactions
Severity	**High**
Description	Currently, there is no way for the SafeGuard to execute emergency transactions. Once the SafeGuard has been set and a timelock period has been configured, all tx (except calls to unlock() must go through the timelock period. This can be limiting and even problematic depending on the purpose of the Safe, for example, if it is used for upgrades or emergency admin of other contracts, not being able to execute txs immediately can lead to catastrophic consequences, specifically in scenarios of ongoing exploits.
Recommendations	Consider adding the possibility to pass a TX if all owners have signed the corresponding TX, similar to the unlock condition.
Comments / Resolution	Resolved, the checkTransaction function has been adjusted to allow immediate executions of unanimous signed transactions.

Issue_03	Lack of sanity check during deployment can result in permanent lock of the Safe
Severity	**High**
Description	During resolution 1, the following additional checks have been implemented within the checkAfterExecution function: *{* *uint256 ownerCount = _safe.ownerCount();* *if (ownerCount < _MINIMUM_OWNERS) {* *revert NotEnoughOwners(ownerCount);* *}* *}* *{* *uint256 threshold = _safe.getThreshold();* *if (threshold < _MINIMUM_THRESHOLD) {* *revert ThresholdTooLow(threshold);* *}* *}* These checks are however not enforced during the contract deployment which means it will be theoretically possible that the SafeGuard is deployed for the Safe while the ownerCount is 1. It will now become impossible to execute any tx from the Safe as the sanity check will always revert (even if one more owner is added, this will bring it to 2 owners and still revert).
Recommendations	Consider adding these sanity checks to the constructor as well. It is important to emphasize that resolution rounds are time-constrained rounds which do not represent a full audit round and thus allocation is limited. The introduction of substantial changes will require a new audit of the said.
Comments / Resolution	

Issue_04	Malicious signer can prevent unlocking the SafeGuard
Severity	**Medium**
Description	The contract exposes a lock feature which can be invoked in case of emergencies. More information on that feature can be found in the corresponding appendix. It is trivially possible for one signer to deny the signature of the unlock call which will then result in the SafeGuard and corresponding Safe to be permanently locked, which is possible due to the fact that this transaction needs unanimous signatures to pass. This only works if a malicious signer is being added as a Safe owner. If the key is compromised, the original owner can simply sign the transaction. This issue has been rated as medium because such a case is rather rare, it would need an effective mismanagement of one of the owners which can be prevented if best-practices are followed(storing key on paper, keeping more than one copy, ...)
Recommendations	There is no trivial fix for this issue besides simply storing all keys properly to ensure all keys for all owners are always retrievable.
Comments / Resolution	Acknowledged, while it would be theoretically possible to enforce signing the _removeOwnerTxHash for multiple subsequent nonces, the malicious actor can always backrun the unlock call by calling lockDown again.

Issue_05	Timelock passed TX cannot be canceled
Severity	**Medium**
Description	The cancel function allows the cancellation of a TX as long as it has not passed the timelock. If the timelock has passed, it will simply revert: *if (block.timestamp > _timelockEnd) {* *revert TimelockElapsed(txHash, _timelockEnd);* *}* Only because the timelock has passed, that does not automatically make the transaction non-malicious. It can indeed happen that the timelock for a malicious TX has passed but it has not yet been executed by the attacker. In such a scenario it should be allowed to cancel the TX. It would still be possible to lock the contract but cancellation is considered the path of the least resistance. On top of that, even if the contract is locked, the attacker can backrun the unlock TX and still execute the TX (in case there are more than 1 TXs (with the same intent) enqueued with subsequent nonces, since the unlock call will increment the nonce and make the previous TX invalid, the subsequent malicious TX can still be triggered). Thus, in such a scenario it becomes mandatory to cancel the TX.
Recommendations	Consider allowing the cancellation even if a TX has passed the timelock.
Comments / Resolution	Acknowledged.

Issue_06	Lack of initial delay setting
Severity	**Medium**
Description	The delay variable determines how long from the enqueue execution it takes until a TX can be executed. This value is set via the setDelay function but is initially zero. In the scenario where a malicious party gains control over the majority of keys shortly after deployment and before the delay setting, it is possible to immediately execute a queued function.
Recommendations	Consider setting the delay in the constructor or defining a value in the storage slot towards it.
Comments / Resolution	Acknowledged, this is by design.

Issue_07	resign hash may be unusable
Severity	**Low**
Description	The caller of the cancel and lockDown functions is required to sign the resign hash. For the cancel function just the current nonce is being used and for the lockDown function the caller is required to the subsequent nonce's hash if he has already signed the current nonce's hash. The rationale behind that is to ensure, if cancel was called and lockDown is called subsequently, that the caller always agrees on his own removal post-unlock. This is to combat eventual griefing attacks where the caller of the cancel function calls lockDown after cancelation which then in turn requires the nonce to be increased before the caller can be removed. It is important to note that the Safe.removeOwner function has the following check: `require(ownerCount - 1 >= _threshold, "GS201");`

	which is necessary to ensure the Safe doesn't lock itself. However, it is important to emphasize that under this scenario, it would be possible to sign the tx but never actually execute it as this requirement would revert. For example, if there is such a scenario where threshold is 3 and there are 3 owners (which could e.g. happen if already one malicious owner was removed), it would never be possible for this safeguard to catch.
Recommendations	Consider keeping this scenario in mind and ensuring that always sufficient owners are existent.
Comments / Resolution	

Issue_08	SafeGuard can't be installed as a guard for Safe v.1.4.0
Severity	**Low**
Description	Currently, the SafeGuard does not expose the corresponding ITransactionGuard interface. While this has no definitive impact on V1.3.0 besides of being a lack of best practices, it will result in the inability to use the SafeGuard for the Safe starting from V1.4.0 due to the deployment check: Safe v.1.4.0 requires the Guard to implement the `supportsInterface[]` on the Guard's contract. This breaks the current deployment mechanism for the SafeGuard, which enforces that the SafeGuard has already been set as the guard of the Safe. For Safe v.1.4.0 it won't be possible to set the SafeGuard as the guard before it has been deployed. This issue has been rated as low severity instead of high severity since the contract is only meant to be used with V1.3.0

Recommendations	Consider simply incorporating the interface.
Comments / Resolution	Acknowledged.

Issue_09	Permissionless nature of enqueue increases user flexibility
Severity	**Low**
Description	The enqueue function allows anyone to enqueue a TX as soon as it has sufficient signatures to meet the threshold. While this does not expose any immediate harm, most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider simply adding the onlyOwner modifier to the enqueue function.
Comments / Resolution	Acknowledged.

Issue_10	Hardcoded OWNER_COUNT_SLOT
Severity	**Low**
Description	Within the SafeLib library the OWNER_COUNT_SLOT is hardcoded to 3, which is correct for the standard Safe implementation. However, if the SafeGuard is meant to be used for custom Safe contracts or for a potential future version, this slot may be inaccurate.
Recommendations	Consider implementing a sanity check during the deployment which ensures that the return value of the OWNER_COUNT_SLOT is within a reasonable range.
Comments / Resolution	Acknowledged, it is expected that the slot stays the same.

Issue_11	Lack of reasonable delay value
Severity	**Low**
Description	The setDelay function allows for setting the delay . Currently, there is no validation which ensures that the setting will result in a reasonable value.
Recommendations	Consider setting the delay to a reasonable value.
Comments / Resolution	Acknowledged, the team indicated that the uint24 type is considered as sufficient validation.

Issue_12	Potential UX issues due to insufficient forwarded gas by Safe frontend
Severity	**Informational**
Description	It is likely that the Safe frontend calculates the to provided gas based on the safeTxGas (if gas parameters are non-zero). An indication for this logic can be found within the execTransaction function which ensures sufficient gasLeft() after the external call to emit events. If this scenario occurs, the TX may not have sufficient gas left for the checkAfterExecution call. This is due to the fact that the remaining gas calculation is only for emitted events plus some additional buffer, which will be insufficient to execute the checkAfterExecution function (especially the storage write).
Recommendations	Consider being conscious about this when broadcasting transactions.
Comments / Resolution	Acknowledged.

Issue_13	Redundant safeguard for cancellation
Severity	**Informational**
Description	The cancel function exposes the antiGriefing modifier which requires two states: a) Caller being owner of the safe b) Caller having subsequent unlock TX signed The latter safeguard seems rather redundant for this context.
Recommendations	Since the antiGriefing safeguard is also used for the lock function, we recommend simply acknowledging this issue instead of introducing another additional modifier only for the cancel function.
Comments / Resolution	Resolved, the cancel function now only requires the onlyOwner modifier and an additional anti-griefing measurement which ensures that the caller has signed the _removeOwnerTxHash.

Issue_14	Canceled TX can never be queued again
Severity	**Informational**
Description	Once a TX has been canceled, the exact same TX can never be enqueued again: *if (timelockEnd[txHash] != 0) {* *revert AlreadyQueued(txHash);* *}* While this is in favor of security, it will result in a small performance decrease of a UX, which is in our opinion acceptable.
Recommendations	Consider acknowledging this issue and simply incrementing the nonce if the same TX should be executed.

Comments / Resolution	Acknowledged, this is by design.

Issue_15	Off-by-one error within timelock check
Severity	**Informational**
Description	The checkTransaction function checks the timelock pass as follows: *if (block.timestamp <= _timelockEnd) {* *revert TimelockNotElapsed(txHash, _timelockEnd);* *}* In case block.timestamp = _timelockEnd, the function simply reverts, which is considered an off-by-one error.
Recommendations	Since this is against the favor of a user, we are of the opinion that it can be acknowledged because it indeed increases the security of the contract. In fact, it even prevents atomic executions in the scenario where delay is zero.
Comments / Resolution	Acknowledged, as per recommendation the client confirms this is by design.

Issue_16	Guard removal is irreversible
Severity	**Informational**
Description	Once the guard is removed, it will set the _guardRemoved variable to true within the checkAfterExecution function. Even if the Safe sets the same guard again it will not set _guardRemoved to false, effectively not implementing any safety features.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_17	resignHash that was pre-approved for cancel() when the guard is not locked down won't be returned correctly on the resignTxHash()
Severity	**Informational**
Description	When a resignHash with the current nonce has been pre-approved, the resignTxHash() will always increment the nonce and retrieve a new hash with the incremented nonce. Approved resignHash when canceling a tx is hashed with the current nonce, not the next nonce (unless the guard is locked down)
Recommendations	Consider simply keeping this issue in mind and acknowledging it.
Comments / Resolution	