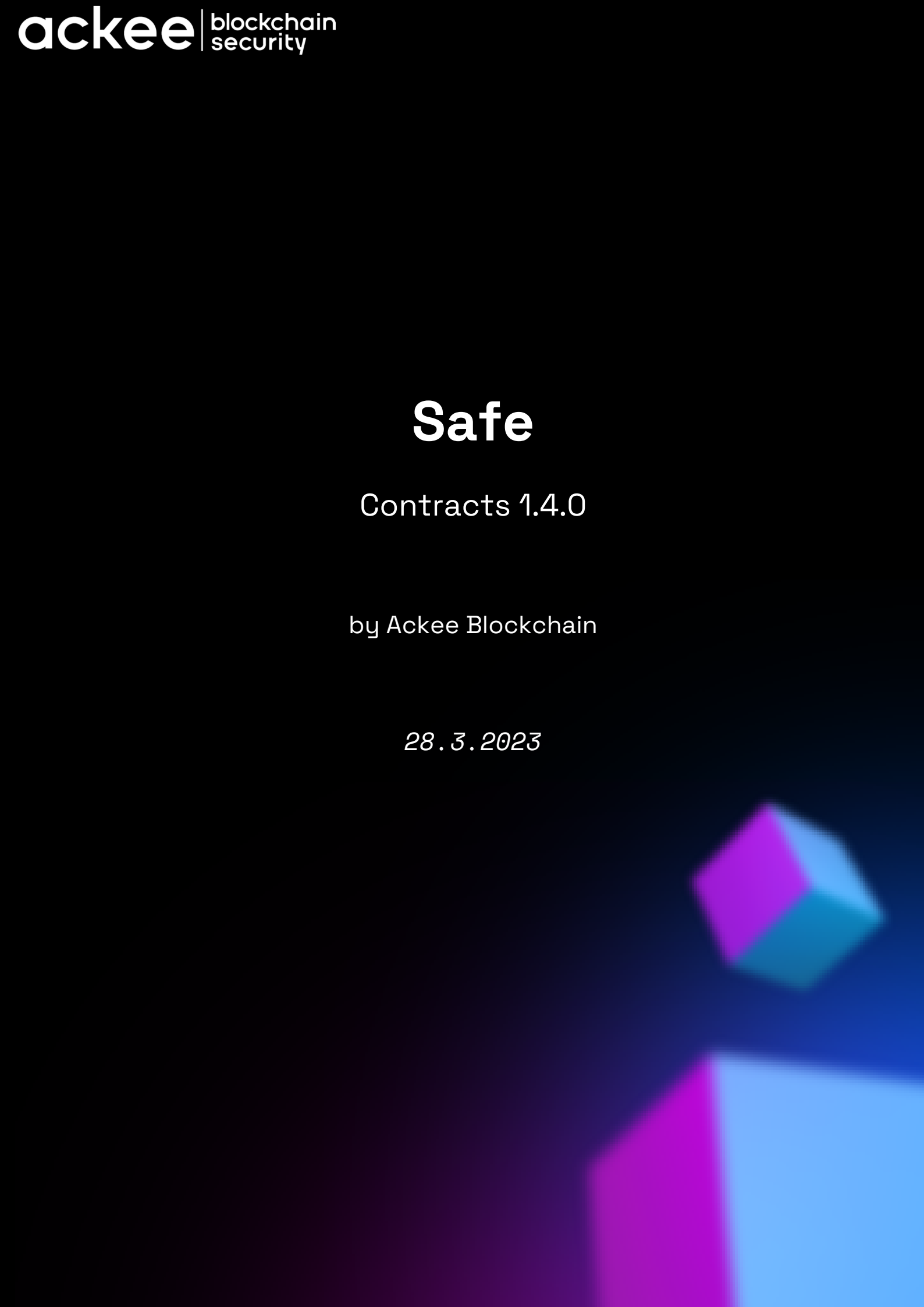
ackee | blockchain security

Safe

Contracts 1.4.0

by Ackee Blockchain

28.3.2023



ackee | blockchain security

Contents

1. Document Revisions

0.1	Draft report	March 14, 2023
1.0	Final report	March 16, 2023
1.1	Fix review	March 28, 2023

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Woke is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzzy testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzzy tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Medium	-
	Low	Medium	Medium	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member's Name	Position
Lukáš Böhm	Lead Auditor
Miroslav Škrabal	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Safe is a decentralized custody protocol allowing multi-signature (multi-sig) wallets to be used as a single account. Businesses and individuals can use multi-sig wallets for safe collective management, perform sensitive transactions, and achieve redundancy. The protocol is widely used across the Ethereum and EVM ecosystems.

Revision 1.0

Safe engaged Ackee Blockchain to perform a security review of the **Safe contracts version 1.4.0** with a total time donation of 10 engineering days in a period between February 27 and March 10, 2023 and the lead auditor was Lukáš Böhm.

The audit has been performed on the commit `eb93dbb`, and the scope was the following contracts with all imports (recursively):

- SafeL2.sol
- proxies/SafeProxyFactory.sol
- handler/CompatibilityFallbackHandler.sol
- libraries/MultiSendCallOnly.sol
- libraries/SignMessageLib.sol

We began our review using static analysis tools Woke. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we have involved Woke testing framework where we simulated deployment of the Safe and focused on the correctness of signature and owner handling. The appendix includes parts of the testing source core.

During the review, we paid particular attention to:

- signature validation,
- malicious owner actions,
- modules handling,
- owners handling,
- guard handling,
- fallback handler logic,
- access controls,
- delegate call risks,
- data validation.

Our review resulted in 11 findings, ranging from Info to Medium severity. The quality of the code is exceptional. NatSpec in-code documentation is part of every contract and function. General documentation still needs to be created, but Safe team provided a few documents describing the most crucial part - signatures.

Ackee Blockchain recommends Safe:

- change guard management logic,
- mitigate impacts of malicious deployer,
- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The review was done on March 28 on the given commit `cb4b2b1`.

The status of all reported issues has been updated and can be seen in the findings table. Issues include client responses, comments, and pull requests

with specific responses, if any.

See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope.

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Solution*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
M1: Broken guard can cause DoS	Medium	1.0	Acknowledged
M2: Lack of contract check	Medium	1.0	Acknowledged
L1: Error-prone proxy constructor	Low	1.0	Acknowledged
W1: Usage of delegatecalls	Warning	1.0	Acknowledged
W2: Fallback handler can be set to address(this)	Warning	1.0	Fixed
W3: Removed owner's stored hash	Warning	1.0	Acknowledged
W4: Singleton address at slot 0	Warning	1.0	Acknowledged

	Severity	Reported	Status
W5: Call to disableModule can be frontrun	Warning	1.0	Acknowledged
W6: Threshold can be set too high	Warning	1.0	Acknowledged
I1: Code and comment inconsistency	Info	1.0	Partially fixed
I2: Require should be assert	Info	1.0	Acknowledged

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts

Contracts we find important for better understanding are described in the following section.

SafeL2

The main logic of the protocol is in the `Safe.sol` contract. It allows execution of the Safe transaction to a specific address, with a data payload, value, and other parameters. All of these parameters must be signed by the owners of the Safe. There are four different signatures:

- Contract signature defined by EIP-1271
- EOA signature
- EIP-712 signature
- Pre-Validated signature

Every signature is 65 bytes long. They are concatenated, sorted by address value in ascending order, and passed as an input parameter into the execution function. In the case of contract signature (EIP-1271), additional data are added to the end of concatenated signatures bytes. The threshold defines the number of signatures required to execute a Safe transaction.

Owners of the Safe are manageable by OwnerManager contract. Transactions can also be executed by modules, and arbitrary transaction guard can be

added. SafeL2 extends the functionality of Safe by emitting events with additional information.

The contract inherits the logic of `./base/xManager.sol` contracts and `./common/x.sol` contracts.

OwnerManager

The contract manages the owners of Safe. It allows adding, removing, and swapping owners. It also allows changing the threshold of the Safe. Functions of the contract are protected by `authorized` modifier. The modifier allows only the Safe contract to call the functions, which means a Safe transaction has to be performed on the Safe contract to call itself.

GuardManager

The contract implements the logic for hooks that are called before and after a Safe transaction is executed. Only one `Guard` can be set at the time. Setting the guard is protected by `authorized` modifier.

ModuleManager

The contract enables and disables modules. These two functionalities are protected by `authorized` modifier. If a module is set, it can execute a Safe transaction without needing signatures to be passed as a parameter into the function.

SafeProxyFactory

The contract is used for deploying the new proxy contract.

- Deployer can choose between 3 different deployment options:
- Deploy Proxy with s nonce
- Deploy chain-specific Proxy with a nonce

- Deploy Proxy with a callback and nonce

The new Proxy is created using the `CREATE2` function, where the address can be precalculated with salt. Singleton Safe contract code is used as a logic contract for the new Proxy.

CompatibilityFallbackHandler

The contract provides compatibility between Safe version < 1.3.0 and 1.3.0 +. It implements the EIP-1271 interface and other necessary functionalities for the new Safe version.

Actors

This part describes the system's actors, roles, and permissions.

Deployer

The deployer of the new Safe has the privilege to set up owners, threshold, and fallback handler.

Owners

Owners of the Safe can sign transactions that are then executable by anyone. The executor of the Safe transaction has to pass all necessary signatures (defined by threshold) as an input argument to the Safe transaction.

Modules

Modules are contracts that can execute Safe transactions without the need for signatures. They can be enabled and disabled by the Safe transaction.

5.2. Trust model

Modules can execute Safe transactions without signature. It is crucial to

carefully select addresses with such trust and privilege over the system.

M1: Broken guard can cause DoS

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Safe.sol	Type:	Denial of service

Description

Safe can set up a guard contract that executes functions before

```
address guard = getGuard();
{
    if (guard != address(0)) {
        Guard(guard).checkTransaction(
            // Transaction info
            ...
        );
    }
}
```

and after transactions.

```
{
    if (guard != address(0)) {
        Guard(guard).checkAfterExecution(txHash, success);
    }
}
```

If one of these two functions is broken or just reverts, it can cause DoS for the whole Safe.

Vulnerability scenario

The guard setup function is protected by `authorized` modifier, thus can be called by Safe transaction only. However, if the guard function reverts for

any reason, there is no way to execute a Safe transaction and change the guard address.

Recommendation

Guard can work as an additional layer of security for the Safe. Nevertheless, if the guard functions contain an issue that causes reverting transactions, Safe should be able to execute transactions without it or have the ability to change the guard address.

Fix 1.1

Client's response:

- *"Modules can be used for recovery"*
- Add documentation for guards and warn about usage

Pull request #535 with added documentation.

Go back to Findings Summary

M2: Lack of contract check

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	SecuredTokenTransfer.sol, Execute.sol	Type:	Data validation

Description

For transferring tokens from the Safe contract to the payment token receiver following function used:

```
function transferToken(address token, address receiver, uint256 amount)
internal returns (bool transferred) {
    // 0xa9059cbb - keccack("transfer(address,uint256)")
    bytes memory data = abi.encodeWithSelector(0xa9059cbb, receiver,
amount);
    // solhint-disable-next-line no-inline-assembly
    assembly {
        // We write the return value to scratch space.
        // See
https://docs.soliditylang.org/en/v0.7.6/internals/layout_in_memory.html#lay
out-in-memory
        let success := call(sub(gas(), 10000), token, 0, add(data, 0x20),
mload(data), 0, 0x20)
        switch returndatasize()
        case 0 {
            transferred := success
        }
        case 0x20 {
            transferred := iszero(or(iszero(success), iszero(mload(0))))
        }
        default {
            transferred := 0
        }
    }
}
```

It uses a low-level call to a predefined function selector `"transfer(address,uint256)"` on the token's address. However, if the address is not a contract, a low-level call returns 1, because no revert happens inside the call.

The second place where contract check is suitable but not performed is the `execute` function in the `Executor` contract.

```
 if (operation == Enum.Operation.DelegateCall) {
    // solhint-disable-next-line no-inline-assembly
    assembly {
        success := delegatecall(txGas, to, add(data, 0x20), mload(data), 0,
0)
    }
} else {
    // solhint-disable-next-line no-inline-assembly
    assembly {
        success := call(txGas, to, value, add(data, 0x20), mload(data), 0,
0)
    }
}
```

The function is called from the Safe execution function, where the check of whether the address **to** is a contract is not performed either. Especially for a delegate call, it is important to check whether the address is a contract.

Vulnerability scenario

Suppose the wrong address is provided as a token or destination address. In that case, the Safe contract will assume that the token transfer or call was successful because the variable `success` is equal to **1**.

Recommendation

Perform the check whether the address is a contract before calling the low-level call.

Additionally, differ the logic for EOA and contract calls.

Fix 1.1

Client's response:

- *"Checks for the execution should be done on interface level. Contracts should provide full flexibility. No user funds at risk"*
- Add documentation that outlines that it is the responsibility of the interface/relayer to check this.

Pull request #536 with added in-code documentation.

Go back to Findings Summary

L1: Error-prone proxy constructor

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	SafeProxy.sol	Type:	Data validation

Description

The constructor of the SafeProxy contract does not use robust verification for singleton address. Only the check for zero address is performed.

Vulnerability scenario

Passing the wrong `singleton` address to the constructor of the SafeProxy contract will lead to the unintended behavior of the contract.

Recommendation

More robust verification can be performed by checking the singletons's identifier against the constant. For example:

```
contract SafeProxy{

    constructor(address _singleton) {
        require(Safe(payable(_singleton)).identifier() == keccak256("safe-1.4.0"), "Invalid singleton address provided");
        singleton = _singleton;

    }
    ...
}

contract Safe{
    function identifier() public pure returns (bytes32) {
        return keccak256("safe-1.4.0");
    ...
```

```
}
```

Fix 1.1

Client's response:

- *"This is intended behavior"*
- *"Proposed solution is not stable enough as it is easily possible to "fake" the check. Alternatively we could fix it to a specific address, but this would defeat the purpose of a generalized proxy."*

Go back to Findings Summary

W1: Usage of delegatecalls

Impact:	Warning	Likelihood:	N/A
Target:	Safe.sol	Type:	Data validation

Description

Delegatecall in setup

The Safe contract uses the `setup` function to initialize its state. The `setup` function can be viewed as a point of centralization as it is called before the owners are set. What is more, the setup function can also result in a delegatecall. That increases the possibility for the deployer to set up the contract dishonestly.

To trust the setup, the owners must verify the code and the inputs to the `setup` function.

The setup process should be as transparent as possible to allow all the parties to verify its output. If a delegate call is used, the probability that everyone will verify the setup is lower.

Delegatecall in execute

The delegatecall can also be triggered from `executeTransaction` or `executeTransactionFromModule`. Such calls are inherently dangerous as they can transform the contract's storage into an inconsistent state. The target contract might not be audited and might break some important invariants (like the owner list validity, nonce linearly decreases, the threshold is at most len(owners) etc.)). If the nonce decreases, transactions might be replayed. If the threshold exceeds the number of owners, the contract might be locked forever, etc.

Recommendation

Include the bare minimum of logic in the setup function. If a more delicate setup is needed, consider moving it to the execute portion of the contract. The delegatecall may be eventually needed, but splitting the setup into two parts makes the verification process more transparent.

More generally, consider the usage of delegatecalls. The semantics can often be easily replicated with a simple call, which is easier to verify and audit.

Go back to Findings Summary

W2: Fallback handler can be set to address(this)

Impact:	Warning	Likelihood:	N/A
Target:	FallbackManager.sol	Type:	Data validation

Description

The fallback handler in the contract FallbackManager can be set to `address(this)` by Safe. It could bypass the Safe's `authorized` modifier in exceptional cases.

The `authorized` modifier enforces a self-call. The fallback handler contains the following code:

```
// The msg.sender address is shifted to the left by 12 bytes to remove the padding
// Then the address without padding is stored right after the calldata
mstore(calldatasize(), shl(96, caller()))
// Add 20 bytes for the address appended add the end
let success := call(gas(), handler, 0, 0, add(calldatasize(), 20), 0, 0)
```

If the `msg.sender` is a specially crafted address, the first 4 bytes of the address may correspond to some function selector. It could be called if such a function has the `authorized` modifier.

So even though the first call will fall into the fallback function, the second one might not.

In the current implementation, this should not be possible because even if the selector matched, the call would revert on `abi.decoding` of the arguments (because the caller's address does not constitute proper calldata for such a function call).

Recommendation

Ensure the fallback handler cannot be set to `address(this)`. This will not reduce the functionality of the fallback handler and will ensure that the handler cannot be set to `address(this)` by accident.

Fix 1.1

Client's response:

- Add `require(handler != this)`
- Add test
- Add in-code documentation

Pull request #534 with a complete fix.

Go back to Findings Summary

W3: Removed owner's stored hash

Impact:	Warning	Likelihood:	N/A
Target:	Safe.sol	Type:	Redundant memory

Description

The Safe provides to approve specific messages by owners.

```
function approveHash(bytes32 hashToApprove) external {
    require(owners[msg.sender] != address(0), "GS030");
    approvedHashes[msg.sender][hashToApprove] = 1;
    emit ApproveHash(hashToApprove, msg.sender);
}
```

However, when one of the owners is removed, the hash of the message he/she approved before remains stored. This fact violates the condition that only owners can make message approvals.

Recommendation

Even though the pre-approved message hashes are not exploitable, there is no reason to store hashes of the removed owner. Therefore, the hash of the removed owner should be removed from the storage.

Fix 1.1

Client's response:

- Add documentation that hashes stay valid

Pull request #538 with added in-code documentation.

Go back to Findings Summary

W4: Singleton address at slot 0

Impact:	Warning	Likelihood:	N/A
Target:	SafeProxy.sol	Type:	Proxy pattern

Description

The SafeProxy contract uses the proxy pattern to delegate calls to the logic contract. The address of the logic contract is stored at slot 0.

```
contract SafeProxy {
    // Singleton always needs to be first declared variable, to ensure that
it is at the same location in the contracts to which calls are delegated.
    // To reduce deployment costs this variable is internal and needs to be
retrieved via `getStorageAt`
    address internal singleton;
```

This is prone to error as it requires that the singleton variable is always the first declared variable in the contract. If not, a slot collision can happen, and the address can get overwritten.

Recommendation

If there are no compatibility issues (like upgrading between different safe versions), the unstructured storage pattern should be performed, as it is much less prone to errors. In the long term, the unstructured storage pattern should be preferred for developing new contracts.

The OpenZeppelin contracts follow this pattern:

```
bytes32 private constant implementationPosition = bytes32(uint256(
  keccak256('eip1967.proxy.implementation')) - 1
));
```

More information can be found in the [OpenZeppelin documentation](https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies#unstructured-storage-proxies).

Go back to Findings Summary

W5: Call to disableModule can be frontrun

Impact:	Warning	Likelihood:	N/A
Target:	Safe.sol, ModuleManager.sol	Type:	Frontrunning

Description

Modules can be added to the Safe and removed. Removing a module is done by calling the `disableModule` function. However, the disabled transaction can be front-run by a malicious module. Because the module can perform state changes in the Safe it also can entirely mitigate the effect of the `disableModule` call.

Recommendation

This issue cannot be mitigated as it is inherent to the Safe design. The issue is included to demonstrate further the potential dangers of using modules.

Fix 1.1

Client's response:

- *"As mentioned in the report modules are "omnipotent" and need to be considered carefully, therefore this should be considered when developing a protocol around the Safe core contracts that provides security."*
- Improve Module documentation

Pull request #539 with added documentation.

Go back to Findings Summary

W6: Threshold can be set too high

Impact:	Warning	Likelihood:	N/A
Target:	OwnerManager.sol	Type:	Data validation

Description

The 5.1.1.2 contract allows adding new owners and changing the threshold. The threshold can be set to arbitrarily high values if it is lower than the number of owners.

However, there is an implicit limit for the threshold imposed by the block gas limit. If the threshold is set too high, supplying enough signatures will not be possible because of the gas limit.

We ran some back-of-the-envelope calculations and found that the threshold would have to reach unreasonable values before the gas limit would become a problem, and thus this should be fine for any multi-sig.

Recommendation

Consider performing some more thorough calculations and setting a limit for the threshold.

Fix 1.1

Client's response:

- *"This is highly dependent on the chain and the limits on the chain. Even on mainnet the limit is quite high therefore we don't see a need for an immediate action (beyond sharing it as part of the audit)"*

Go back to Findings Summary

I1: Code and comment inconsistency

Impact:	Info	Likelihood:	N/A
Target:	Safe.sol, ModuleManager.sol	Type:	Code quality

Description

While declaring new variables in Safe contract, at the line **#150** a zero value is assigned,

```
uint256 moduleCount = 0;
```

at the line **#276** it is not.

```
uint8 v;
```

Even though the compiler assigns a zero value to the variables, it is a good practice not to mix the two approaches.

In the contract ModuleManager the code comment at line **#160** refers to the variable `currentModule`, which does not exist in the code.

```
the `currentModule` will always be either a module address
```

Recommendation

Stick with one approach for an assignment and use it consistently across the codebase.

Update the in-code comment to refer to the correct variable.

Fix 1.1

Client's response:

- Variable declaration: *"No action because in our tests leaving a variable non-initialized resulted in less gas consumption"*
- The comment in `ModuleManager` was adjusted

Pull request #530 with adjusted in-code documentation.

Go back to Findings Summary

I2: Require should be assert

Impact:	Info	Likelihood:	N/A
Target:	OwnerManager.sol, ModuleManager.sol	Type:	Code quality

Description

The `require` statement is used instead of better suited `assert` at several places in the code. The `require` statement checks conditions that are not supposed to happen during regular operation. However, the `assert` statement checks conditions that should always be true.

The following `require` statements should be `asserts` :

- OwnerManager.sol
  - `setupOwners` #31

```
require(threshold == 0, "GS200");
```

- ModuleManager.sol
  - `setupModules` #32

```
require(modules[SENTINEL_MODULES] == address(0), "GS100");
```

These invariant conditions should always be true and are not supposed to happen during regular operations.

It is essential to remember that solidity version `< 0.8.0` (allowed version for Safe contracts) failing `asserts` are returning **invalid** opcode, which consumes all remaining gas. On the other hand, `require` is returning unused gas.

Recommendation

The `asserts` provide more information for reviewers and auditors because they convey that the given condition should always be true. Using `requires` may be confusing because it implies that the condition could sometimes revert.

Fix 1.1

Client's response:

- *"The current version of the Safe uses Solidity 0.7.6 where require does not use up all gas. To prevent unexpected behavior this will not be changed in this version"*

Go back to Findings Summary

6. Report revision 1.1

No significant changes were performed in the logic of contracts. Events were modified in several places in the codebase (PR #542). They are now indexed for better off-chain access. All other changes address reported issues.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Safe: Contracts 1.4.0, 28.3.2023.

Appendix B: Woke outputs

B.1. Tests

The following code shows the functions implemented in Woke testing framework for building every type of signature payload that is used in the Safe contract and also `create_mutlisig` function for creating the final signature byte payload.

```
# Static part v == 0
def get_eip_sig(address, offset):
    # r - contract address
    contract = 12 * b"\x00" + bytes.fromhex(str(address)[2:])
    # s - pointing to dynamic data start
    data_pointer = int.to_bytes(offset, 32, "big")
    # v - type
    sig_type = b"\x00"
    static_part = contract + data_pointer + sig_type
    return static_part

# Dynamic part v == 0
def get_eip_dynamic_data(data):
    # 32 bytes - len of following data
    data_len = int.to_bytes(len(data), 32, "big")
    # len + data
    dynamic_part = data_len + data
    return dynamic_part

# v > 30 branch - formated EIP-712 msg
def get_formated_sig(address, hash):
    sig = address.sign(hash)
    r = sig[:32]
    s = sig[32:64]
    v = sig[64:]
    signature = r + s + int.to_bytes(int.from_bytes(v, "big") + 4, 1,
"big")
    return signature

# default sig
```

```
def get_classic_sig(address, hash):
    signature = address.sign_hash(hash)
    return signature

# v == 1 branch - owner in r, the rest does not matter
def get_r_owner_payload(address):
    signature = 12 * b"\x00" + bytes.fromhex(str(address)[2:]) + 32 *
b"\x00" + b"\x01"
    return signature

# Create final signature with ascending order of addresses (magic)
def create_mutlisig(v_0, v_1, v_30, default, hash, data):
    # Create tuples (address, type_of_signature)
    joined =[]
    if v_0:
        v_0_tup = create_tuple(v_0,'0')
        joined += v_0_tup
    if v_1:
        v_1_tup = create_tuple(v_1,'1')
        joined += v_1_tup
    if v_30:
        v_30_tup = create_tuple(v_30,'30')
        joined += v_30_tup
    if default:
        default_tup = create_tuple(default,'def')
        joined += default_tup
    # sort it
    joined.sort(key=lambda tup: tup[1])
    # final mutlisig bytes
    multisig = b""
    # data bytes of EIP sig - connect at the end of static signatures
    eip_data = b""
    # if more EIP sigs (v==0) - we need to point to the end of data in
memory
    multisig_len = len(joined) * 65
    for tup in joined:
        if tup[0] == '0':
            multisig += get_eip_sig(tup[1], multisig_len)
            eip_data += get_eip_dynamic_data(data)
            multisig_len += len(eip_data)
        elif tup[0] == '1':
            multisig += get_r_owner_payload(tup[1])
```

```
        elif tup[0] == '30':
            multisig += get_formated_sig(tup[1], hash)
        else:
            multisig += get_classic_sig(tup[1], hash)

    multisig += eip_data
    return multisig

def create_tuple(v, id):
    tup = []
    for a in v:
            tup.append((id,a))
    return tup
```

Initial deployment code of the Safe by SafeProxyFactory.

```
# Complete safe deployment
def setup_safe(owners, treshold, handler, token, receiver):
    deployer = owners[0]
    # Singleton original Safe contract
    singleton   = Safe.deploy(from_=deployer)
    factory     = SafeProxyFactory.deploy(from_=deployer)
    # Proxy take singleton code and call create2
    proxy = factory.createProxyWithNonce(
        singleton,
        b"",
        42,
        from_=deployer
    )
    # Calling our Safe contract methods trough proxy address
    safe = Safe(proxy.address)
    safe.setup(
        owners,
        treshold,
        Address(0),     # no modules
        b"",            # no data
        handler,        # fallback handler (address)
        token,          # payment token (address)
        0,              # payment
        receiver,       # payment receiver (address)
        from_=deployer,
    )
    return safe
```

Example of guard setup Safe transaction with provided signatures:

```
    # Default sig
    a = Account.from_alias("test")
    a.balance = 10*(10**18)
    # V > 30 sig
    b = default_chain.accounts[1]
    c = default_chain.accounts[2]
    d = default_chain.accounts[3]
    e = default_chain.accounts[4]
    # V == 0 sig
    contract_1 = SignatureValidator.deploy(from_=c)
    contract_2 = SignatureValidator.deploy(from_=e)
    # 0 < treshold <= len(owners)
    owners = [a,b,contract_1,d,contract_2]
    treshold = 5

    ...

    guard = DebugTransactionGuard.deploy(from_=c)
    tx_abi = Abi.encode_call(Safe.setGuard, [guard.address])
    ###### SETTING GUARD ######
    to = safe.address
    value = 0
    data = tx_abi
    operation = Enum.Operation.Call
    safe_tx_gas = 100000
    base_gas = 100000
    gas_price = 0
    nonce = 0

    tx_data = safe.encodeTransactionData(
        to,
        value,
        data,
        operation,
        safe_tx_gas,
        base_gas,
        gas_price,
        payment_token,
        payment_receiver,
```

```
        nonce,
        from_=d
    )

    tx_hash = keccak256(tx_data)
    contract.sign(tx_hash, b"\x00",from_=c)
    contract_2.sign(tx_hash, b"\x00",from_=e)

    # Packed byte signatures in 'multisig' var
    # v_0   - array of addresses where contract is a signer
    # v_1   - array of addr. with approved hashes
    # v_30  - array of addr. for eth signed messages
    # def   - array of addr. for classic signatures
    multisig = create_mutlisig(
        v_0     = [Account(contract_1.address),
                   Account(contract_2.address)],
        v_1     = [d],
        v_30    = [b],
        default = [a],
        hash    = tx_hash,
        data    = tx_data
    )

    tx = safe.execTransaction(
        to,
        value,
        data,
        operation,
        safe_tx_gas,
        base_gas,
        gas_price,
        payment_token,
        payment_receiver,
        multisig,
        from_=d,
        return_tx=True
    )
```

ackee | blockchain security

Thank You

Ackee Blockchain a.s.


Prague, Czech Republic


hello@ackeeblockchain.com


https://twitter.com/AckeeBlockchain