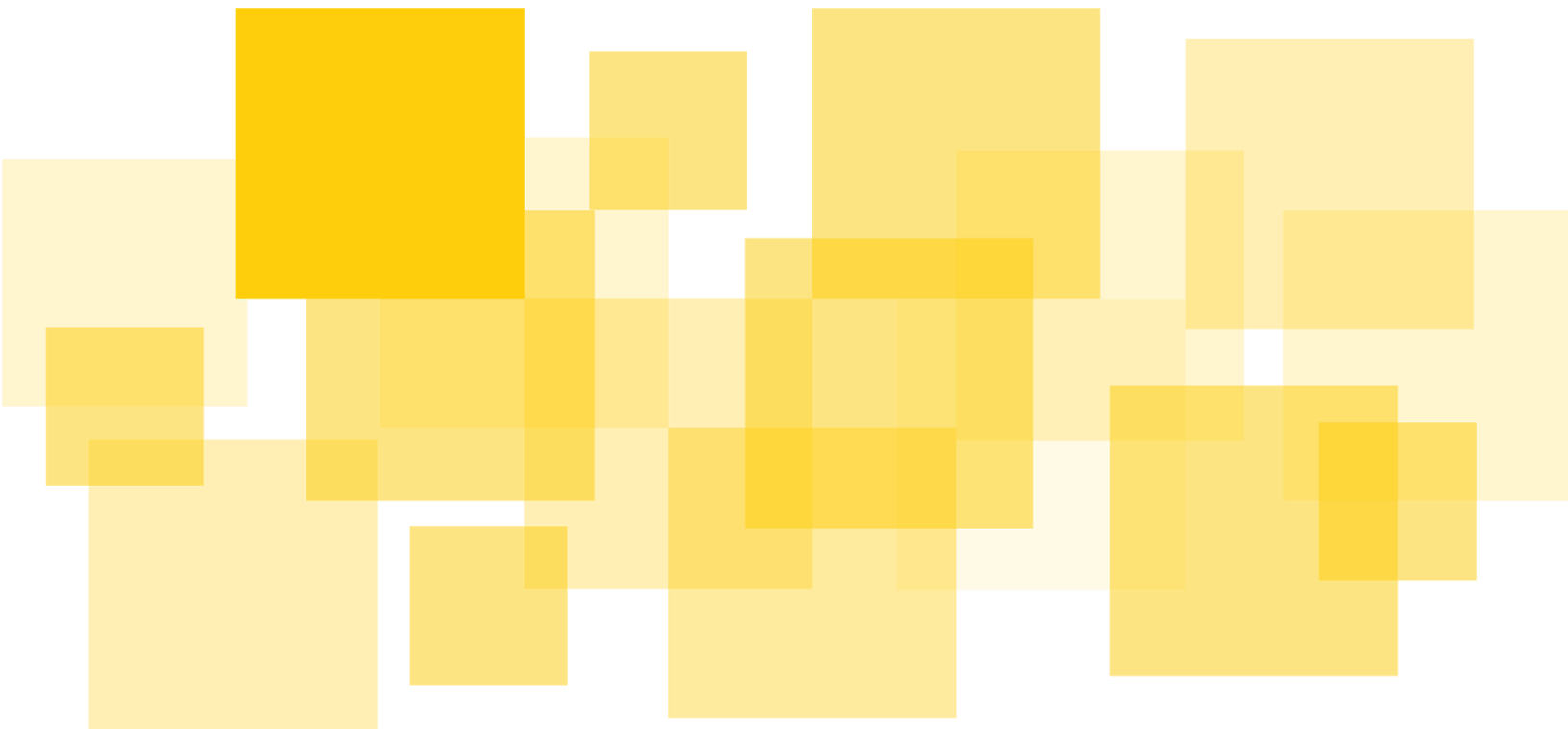


Formal Verification Report

GnosisSafe Contract

Delivered: February 7th, 2019

Updated: February 27th, 2019



Prepared for Gnosis Ltd. by





Table of Contents

Goal & Scope

Disclaimer

Security Audit Overview & Methodology

List of Findings

- Reentrancy vulnerability in `execTransaction`

- `ISignatureValidator` gas and refund abuse

- Transaction reordering vulnerability in `addOwnerWithThreshold`, `removeOwner`, and `changeThreshold`

- `execTransaction` allows a user transaction to the zero address

- `execTransaction` is missing the contract existence check for the user transaction target

- `changeMasterCopy` is missing contract existence check

- Potential overflow if contract invariant is not met

- Potential list index out of bounds in `signatureSplit`

- Missing well-formedness check for signature encoding in `checkSignatures`

Informative Findings & Recommendations

- Lazy enum type check

- Address range

- Scanning `isValidSignature` when adding an owner

- Local validity check of `checkSignatures`

- No explicit check for the case $2 \leq v \leq 26$ in `checkSignatures`

- `handlePayment` allows to send Ether to the precompiled contract addresses

- Insufficient external call result check and gas efficiency of `transferToken`



Common Antipattern Analysis

Formal Specification & Verification Overview

Formal Verification Methodology

Resources

Mechanized Specification and Proof

Formal Specification Details

Assumptions

GnosisSafe contract

Function signatureSplit

Function encodeTransactionData

Function handlePayment

Function checkSignatures

Function execTransaction

OwnerManager contract

Function addOwnerWithThreshold

Function removeOwner

Function swapOwner

ModuleManager contract

Function enableModule

Function disableModule

Function execTransactionFromModule

MasterCopy contract

Function changeMasterCopy

Executive Summary

GnosisSafe is a smart contract that implements a multisignature wallet, supporting various types of signature validation schemes, including ECDSA, [EIP-1271](#), and a contract-builtin approval scheme.

Runtime Verification, Inc. (RV), audited the code and formally verified security-critical properties of the GnosisSafe contract. The set of properties were carefully identified by the Gnosis team, and we faithfully formalized and verified these properties *at the EVM bytecode level*. The formal specification is mechanized within and automatically verified by our EVM verifier, a correct-by-construction deductive program verifier derived from KEVM and K-framework's [reachability logic theorem prover](#).

The formal verification process guided us to systematically reason about all corner cases of the contract, which led us to find several issues, including reentrancy and transaction reordering vulnerabilities, and usability issues that any client of this contract should be aware of. Please note, however, that the vulnerabilities identified are exploitable in rather limited circumstances, where part of the contract owners are required to be malicious and/or compromised.

Update (as of February 27th, 2019): The Gnosis team has [updated](#) their contract following our recommendations for the most critical issues.

Goal & Scope

The goal of the engagement was to audit the code and formally verify security-critical properties of the GnosisSafe contract. RV formally specified the security properties and verified them against the GnosisSafe contract bytecode using the KEVM verifier. The code is from commit ID [427d6f7e779431333c54bcb4d4cde31e4d57ce96](#) of the [gnosis/safe-contracts](#) Github repository.

The scope of the formal verification is the GnosisSafe contract without enabling any add-on modules. Specifically, this includes the following functions:

- executeTransaction of GnosisSafe.sol:
 - only for the case of operation == CALL.
 - including encodeTransactionData, checkSignatures, and handlePayment functions.
- changeMasterCopy of MasterCopy.sol
- addOwner, removeOwner, and swapOwner of OwnerManager.sol
- enableModule, and disableModule of ModuleManager.sol
- execTransactionFromModule of ModuleManager.sol
 - only for the case that modules is empty.

The formal verification is limited in scope within the boundary of the Solidity contract only. Off-chain and client-side portions of the codebase are *not* in the scope of this engagement. See our [Disclaimer](#) next.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

The formal verification results presented here only show that the target contract behaviors meet the formal (functional) specifications, under appropriate assumptions. Moreover, the correctness of the generated formal proofs assumes the correctness of the specifications and their refinement, the correctness of [KEVM](#), the correctness of the [K-framework's reachability logic theorem prover](#), and the correctness of the [Z3](#) SMT solver. The presented results make no guarantee about properties not specified in the formal specification. Importantly, the presented formal specifications consider only the behaviors within the EVM, without considering the block/transaction level properties or off-chain behaviors, meaning that the verification results do not completely rule out the possibility of the contract being vulnerable to existing and/or unknown attacks. Finally, Runtime Verification formally verifies the EVM bytecode and *not* the Solidity source code. Consequently, verification results only apply to a specific EVM bytecode provided by the customer, which we explicitly reference. In particular, modifying/upgrading the Solidity compiler may require all the proofs to be re-executed and the formal specifications modified.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Security Audit Overview & Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in [Disclaimer](#), we have followed the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation, which led us to find, e.g., the issue [#2](#). Second, we carefully checked if the code is vulnerable to known security issues and attack vectors¹, which led us to find, e.g., the issues [#1](#), [#3](#), and [#5](#). Third, we symbolically executed the EVM bytecode of the contract to systematically search for unexpected, possibly exploitable, behaviors at the bytecode level, that can result from quirks or bugs in the EVM or the Solidity compiler itself. This process led us to find, e.g., the issues [#8](#) and [#9](#).

¹ To faithfully identify such known security vulnerabilities, we have consulted various literature on smart contract security including common issues compiled by [ConsenSys](#) and [Sigma Prime](#), and other security audit reports provided by [Philip Daian](#), [Trail of Bits](#), and [Chain Security](#), in addition to the experience of our RV team of auditors and formal methods engineers.

List of Findings

Critical

1. Reentrancy vulnerability in `execTransaction`
2. `ISignatureValidator` gas and refund abuse
3. Transaction reordering vulnerability in `addOwnerWithThreshold`, `removeOwner`, and `changeThreshold`
4. `execTransaction` allows a user transaction to address 0 (zero)
5. `execTransaction` missing the contract existence check for the user transaction target
6. `changeMasterCopy` missing contract existence check
7. Potential overflow if contract invariant is not met
8. Potential list index out of bounds in `signatureSplit`
9. Missing well-formedness check for signature encoding in `checkSignatures`

Informative (non-critical):

10. Lazy enum type check
11. Address range
12. Scanning `isValidSignature` when adding an owner
13. Local validity check of `checkSignatures`
14. No explicit check for the case $2 \leq v \leq 26$ in `checkSignatures`
15. `handlePayment` allows to send Ether to the precompiled contract addresses
16. Insufficient external call result check and gas efficiency of `transferToken`
17. `addOwnerWithThreshold` in case of contract invariant being not satisfied
18. signatures size limit

Reentrancy vulnerability in `execTransaction`

To protect against reentrancy attacks, GnosisSafe employs storage field `nonce`, which is incremented during each transaction. However, there are 3 external calls performed during a transaction, which all have to be guarded from reentrancy.

Below is the code for `execTransaction`, the main function of GnosisSafe:

```
function execTransaction(
    address to,
    uint256 value,
    bytes calldata data,
    ...
    bytes calldata signatures
)
    external
    returns (bool success)
{
    uint256 startGas = gasleft();
    bytes memory txHashData = encodeTransactionData(to, value, data, ..., nonce);
    require(checkSignatures(keccak256(txHashData), txHashData, signatures, true),
        "Invalid signatures provided");
    // Increase nonce and execute transaction.
    nonce++;
    require(gasleft() >= safeTxGas, "Not enough gas to execute safe transaction");
    success = execute(to, value, data, ...);
    if (!success) {
        emit ExecutionFailed(keccak256(txHashData));
    }
    if (gasPrice > 0) {
        handlePayment(...);
    }
}
```

The main external call managed by this transaction (hereafter referred as "payload") is performed in function `execute`. After payload is executed, the original caller or another account specified in transaction data is refunded for gas cost in `handlePayment`. Both

these calls are performed after the nonce [is incremented](#). Consequently, it is impossible to execute the same transaction multiple times from within these calls.

However, there is one more external call possible inside [checkSignatures](#) phase, which calls [an external contract](#) managed by an owner to validate the signature using [EIP-1271](#) signature validation mechanism:

```
function checkSignatures(bytes32 dataHash, bytes memory data,
                        bytes memory signatures, bool consumeHash)

    public
    returns (bool)
{
    for (i = 0; i < threshold; i++) {
        (v, r, s) = signatureSplit(signatures, i);
        // If v is 0 then it is a contract signature
        if (v == 0) {
            // When handling contract signatures the address of the contract
            // is encoded into r
            currentOwner = address(uint256(r));
            bytes memory contractSignature;
            assembly {
                // The signature data for contract signatures is appended to the
                // concatenated signatures and the offset is stored in s
                contractSignature := add(add(signatures, s), 0x20)
            }
            if (!ISignatureValidator(currentOwner)
                .isValidSignature(data, contractSignature)) {
                return false;
            }
        } else { ... }
        ...
    }
    return true;
}
```

This call is performed BEFORE nonce is incremented [here](#), thus remains unprotected from reentrancy.

An owner using EIP-1271 signature validation may use this vulnerability to run the same payload multiple times, despite its approval by other owners to run only once. The limit of how many times a transaction can run recursively is given by call gas and block gas limit, thus the malicious owner will call this transaction with a great deal of gas allocated. The most likely beneficiary of this attack is the owner who initiated the transaction. Yet if a benign owner calls another malicious contract for the signature validation, the malicious contract can exploit said contract even if he is not an owner.

Exploit Scenario

Suppose we have a Gnosis safe managed by several owners, which controls access to an account that holds ERC20 tokens. At some point they agree to transfer X tokens from the safe to the personal account of owner 1.

Conditions required for this attack are detailed below:

- (a). Owner 1 is a contract that uses [EIP-1271](#) signature validation mechanism.
- (b). All other owners use either EIP-1271 or ECSDA signatures. (See [this page](#) for the 3 types of signature validation.)
 - 1. Owner 1 generates the transaction data for this transfer and ensures that allocated gas is 10x required amount to complete the transaction.
 - 2. Owner 1 requests signatures for this transaction from the other owners.
 - 3. Owner 1 registers a malicious ISignatureValidator contract into his own account, that once invoked, will call the Gnosis Safe with the same call data as long as there is enough gas, then return true.
 - 4. Owner 1 generates a signature for the transaction, of type [EIP-1271](#), e.g. it will call the ISignatureValidator.
 - 5. Owner 1 calls the Gnosis Safe with the transaction data and all the signatures.
 - 6. During signature verification phase, Gnosis Safe invokes the malicious ISignatureValidator, that successfully calls the safe again with the same data, recursively, 9 more times.
 - 7. Owner 1 receives into his account 10X the amount of tokens approved by the other owners.

Recommendation

Increment nonce before calling checkSignatures.

ISignatureValidator gas and refund abuse

The account that initiated the transaction can consume large amounts of gas for free, unnoticed by other owners, and possibly receive a refund larger than the amount of gas consumed.

The attack is possible due to a combination of factors.

First, GnosisSafe emits a refund at the end of transaction, for the amount of gas consumed. The target of the refund is either transaction initiator `tx.origin` (by default) or some other account given by transaction parameter `refundReceiver`. This currency of the refund may either be Ether by default, or an ERC20 token with a specified price per unit. Refund token is given by transaction parameters `gasPrice`, `gasToken`. All the transaction parameters must be signed by the required amount of owners, just like the payload.

The second factor is that gas allocated for the whole `execTransaction` is not part of transaction data. (Yet gas for payload is, as we show below.)


This refund mechanism may in principle be abused because the transaction initiator may spend a large amount of gas without the knowledge of other owners and as a result be refunded. The original owner may receive a benefit from such abuse in the case where (1) the refund is emitted in token, and (2) the gas price in token is greater than the market price of Ether of that token. The latter is plausible, for example because: (1) the gas price is outdated, (2) the market price of token changed following its initial valuation, and (3) owners did not care to adjust the gas price because gas consumption was always small and thus irrelevant.

We again need to analyze the situation on all 3 external call sites. For the payload external call, gas is limited by transaction parameter `safeTxGas`. This parameter must be set and validated by other owners when token refund is used. As a result, abuse is impossible. For the external call that sends the refund in token, gas is limited to remaining gas for transaction minus 10000 [source](#):

```
let success := call(sub(gas, 10000), token, 0, add(data, 0x20), mload(data), 0, 0)
```

This appears to resemble a poor limit, but in order to be abused, the transaction initiator must have control over the token account, which looks like an unlikely scenario.

The biggest concern is again the call to [ISignatureValidator](#). This call is under the control of transaction initiator, and the gas for it is not limited (see code for `checkSignatures`).



Thus, the attacking owner may use a malicious `ISignatureValidator` that consumes almost all allocated gas, in order to receive a large refund. The amount of benefit received by the attacker is limited by (1) block gas limit and (2) ratio between `gasPrice` and market cost of the token. However, we should allow for the possibility that block gas limit will increase in the future. Consequently, this remains a valid vulnerability.

Note that careful gas limits on external contract calls are a common security practice. For example when Ether is sent in Solidity through `msg.sender.send(ethAmt)`, gas is automatically limited to [2300](#).

Recommendation

Limit the gas when calling `ISignatureValidator` to a small predetermined value, carefully chosen by considering the specific functionality of `ISignatureValidator`.

Transaction reordering vulnerability in addOwnerWithThreshold, removeOwner, and changeThreshold

The addOwnerWithThreshold function allows an update to threshold, for which a race condition exists similarly to the [ERC20 approve race condition](#).

A common usage scenario of addOwnerWithThreshold is to add a new owner while *increasing* the threshold value (or at least keeping the value as is). The case of decreasing the threshold value while adding a new owner, is unlikely. If there still exists such a use case, one can split the task into two transactions: add new owner, and decrease threshold. There is little reason to perform two updates atomically.

The removeOwner function has a similar issue.

Exploit Scenario

Suppose there are five owners with threshold = 3. Suppose Alice proposes (in off-chain) two consecutive transactions, addOwnerWithThreshold(o1,4) and addOwnerWithThreshold(o2,5). Suppose, however, the off-chain operator receives two transactions in reverse order, due to network congestion. If the two transactions are approved in the wrong order by the owners, the final threshold value will be 4, even though it should be 5.

Discussion

The exploit scenario requires that the owners approve the off-chain transactions in the wrong order by mistake or deliberately. Note that once the off-chain transactions are approved in the correct order, it is *not* possible for them to be executed (on-chain) in the wrong order even if miners are malicious. This is because the nonce increases linearly and the signature (collected off-chain for approving a transaction) depends on the nonce, which induces the total order of transactions that GnosisSafe ensures to follow.

However, if the linearly increasing nonce scheme is not adhered in a future version of GnosisSafe (e.g., by employing a different nonce scheme), the presented vulnerability is exploitable even if all the owners are benign and perfect (making no mistake).

Recommendation

- Modify addOwnerWithThreshold to prevent from decreasing threshold.
- Modify removeOwner to prevent from increasing threshold.

- Make `changeThreshold` private, and add the safer alternatives, i.e., `increaseThreshold` and `decreaseThreshold`.

execTransaction allows a user transaction to the zero address

execTransaction does not reject the case of to being the zero address 0x0, which leads to an *internal* transaction to the zero address, via the following function call sequence:

- <https://github.com/gnosis/safe-contracts/blob/v0.1.0/contracts/GnosisSafe.sol#L95>
- <https://github.com/gnosis/safe-contracts/blob/v0.1.0/contracts/base/Executor.sol#L17>
- <https://github.com/gnosis/safe-contracts/blob/v0.1.0/contracts/base/Executor.sol#L33>

Unlike a regular transaction to the zero address, which creates a new account, an internal transaction to the zero address behaves the same as other transactions to non-zero addresses, i.e., sending Ether to the zero address account (which indeed exists: <https://etherscan.io/address/0x00>) and executing the code associated to it (which is empty in this case).

Although it is the users' responsibility to ensure correctness of the transaction data, it is possible a certain user may not be aware of the difference between the regular and internal transactions to the zero address. This can result in the user sending transaction data to execTransaction with to == 0x0, all the while expecting the creation of a new account. Because an internal transaction to the zero address succeeds (note that it spends a small amount of gas without the need to pay the G_newaccount (25,000) fee because the zero-address account already exists), it may cause the Ether to remain stuck at 0x0, which could become a serious concern when the user attaches a large amount of Ether as a startup fund for the new account.

Recommendation

Modify execTransaction to revert when to == address(0).

execTransaction is missing the contract existence check for the user transaction target

execTransaction is missing the contract existence check for the user transaction target, which may result in the loss of Ether.

According to the [Solidity document](#):

The low-level functions call, delegatecall and staticcall return true as their first return value if the called account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.

That is, if a client commits a mistake by providing a non-existing target address when preparing a user transaction, the execute function will silently return true when transferring the paid Ether to the non-existing account. The result is a loss of Ether.

However, it is not trivial to check the existence for a non-contract account.

Recommendation

In the short term, add a check for a contract account, e.g., requiring `extcodesize(to) > 0` when data is not empty and operation = Call.

In the long term, differentiate the two types of user transactions, i.e., the external contract call transaction and the simple Ether transfer transaction. Implement the contract existence check for the external contract call transaction. With respect to the Ether transfer transaction, explicitly reference this limitation in the document of execTransaction, and/or implement a certain conservative existence check at the client side to provide a warning message if the given address seems to refer to a non-existing account.



changeMasterCopy is missing contract existence check

changeMasterCopy is missing the contract account existence check for the new master copy address. If the master copy is set to a non-contract account, the Proxy fall-back function will silently return.

Recommendation

Implement the existence check, e.g., `extcodesize(_masterCopy) > 0`.

Potential overflow if contract invariant is not met

There are several places where SafeMath is not employed for the arithmetic operations.

- <https://github.com/gnosis/safe-contracts/blob/v0.1.0/contracts/GnosisSafe.sol#L92>
- <https://github.com/gnosis/safe-contracts/blob/v0.1.0/contracts/GnosisSafe.sol#L139>
- <https://github.com/gnosis/safe-contracts/blob/v0.1.0/contracts/base/OwnerManager.sol#L62>
- <https://github.com/gnosis/safe-contracts/blob/v0.1.0/contracts/base/OwnerManager.sol#L79>
- <https://github.com/gnosis/safe-contracts/blob/v0.1.0/contracts/base/OwnerManager.sol#L85>

The following contract invariants are necessary to rule out the possibility of overflow:

- `nonce` is small enough to avoid overflow in `nonce++`.
- `threshold` is small enough to avoid overflow in `threshold * 65`.
- `ownerCount >= 1` is small enough to avoid overflow in `ownerCount++`, `ownerCount - 1`, and `ownerCount--`.

In the current GnosisSafe contract, considering the resource limitation (such as gas), it is reasonable to assume the above invariants. Nonetheless, this examination should be repeated whenever the contract is updated.

Recommendation

Use SafeMath for all arithmetic operations.



Potential list index out of bounds in signatureSplit

The signatureSplit function does not check that the index is within the bounds of the signatures list.

In the current GnosisSafe contract, although no out-of-bounds index is passed to the function, it is still possible for a future implementation to make a mistake, thus passing an out-of-bounds index.

Recommendation

Add the index bounds check or explicitly mention the requirement in the document of signatureSplit to prevent violations in future implementations.

Missing well-formedness check for signature encoding in `checkSignatures`

`checkSignatures` does not explicitly check if the signature encoding is valid.

The signature encoding should satisfy the following conditions to be valid:

- When `v` is 0 or 1, the owner `r` should be within the range of `address`. Otherwise, the higher bits are truncated.
- When `v` is 0:
 - The offset `s` should be within the bounds of the `signatures` buffer, i.e., `s + 32 <= signatures.length`. Otherwise, it will read garbage value from the memory.
 - The dynamic signature data pointed by `s` needs to be well-formed:
 - The first 4 bytes needs to denote the size of the dynamic data, i.e., `dynamic-data-size := mload(signatures + s + 32)`. Otherwise, it may try to read a large memory range, causing the out-of-gas exception.
 - The `signatures` buffer needs to be large enough to hold the dynamic data, i.e., `signatures.length >= s + 32 + dynamic-data-size`. Otherwise, it will read some garbage value from the memory.
 - (Optional) Each dynamic data buffer should not be pointed to by multiple signatures. Otherwise, the same dynamic data will be used to check the validity of different signatures.
 - (Optional) Different dynamic data buffers should not overlap.

For a reference, the following checks are inserted in the bytecode by the Solidity compiler for each bytes-type argument.

```
1. CALLDATASIZE >= 4 ? // checks if the function signature is provided
2. CALLDATASIZE >= 4 + 32 * NUM_OF_ARGS
   // checks if the headers of all arguments are provided
3. .... // load static type arguments and checks the range
4. startLOC := CALLDATALOAD(4 + 32 * IDX)
   // suppose the bytes-type argument is given in the IDX-th position
5. startLOC <= 2^32 ?
6. startLOC + 4 + 32 <= CALLDATASIZE ?
   // checks if the length information is provided
7. dataLen := CALLDATALOAD(startLoc + 4)
```

```
8. startLoc + 4 + 32 + dataLen <= CALLDATASIZE ?  
    // checks if the actual data buffer is provided  
9. dataLen <= 2^32 ?  
10. ... CALLDATACOPY(..., startLoc + 4 + 32, dataLen) ...  
    // copy the data buffer to the memory
```

Discussion

The presented vulnerability allows malicious users to control the memory access (i.e., read) pattern. However, we have not yet found any critical exploit against this vulnerability, but we note that it does not necessarily imply the absence of exploits, and it is not a good practice to admit unintended behaviors.

Recommendation

Implement the signature encoding validity check.

Informative Findings & Recommendations

Here we discuss other identified issues of the GnosisSafe contract that are informative, but not necessarily critical. Nevertheless, we highlight them below to ensure the Gnosis team is fully aware of these issues and of their implications.

Lazy enum type check

The `operation` argument value must be with the range of `Enum.Operation`, i.e., `[0,2]` inclusive, and the Solidity compiler is expected to generate the range check in the compiled bytecode. The range check does not appear in the `execTransaction` function, but appears only inside the `execute` function. We have not yet discovered an exploit of this missing range check. However, it could be potentially vulnerable and requires a careful examination whenever the new bytecode is generated.

Address range

All address argument values (e.g., `to`) must be within the range of address, i.e., `[0, 2160-1]` inclusive. Otherwise, the first 96 (= 256 - 160) bits are silently truncated (with no exception). Thus, any client of the function that takes address arguments should check the validity of addresses before passing them to the function.

Scanning `isValidSignature` when adding an owner

It may be considered to scan the `isValidSignature` function whenever adding a new owner (either on-chain or off-chain), to ensure that the function body contains no malicious opcode.

Example:

- Scanner implementation (in Vyper):
https://github.com/ethereum/casper/blob/master/casper/contracts/purity_checker.py
- Scanner usage (on-chain):
https://github.com/ethereum/casper/blob/master/casper/contracts/simple_casper.v.py#L578

Local validity check of checkSignatures

checkSignatures checks only the first threshold number of signatures. Thus, the validity of the remaining signatures is not considered. Also, the entire list of signatures is not required to be sorted, as long as the first threshold number of signatures are locally sorted. However, we have not found any attack exploiting this.

Another questionable behavior is the case where there are threshold valid signatures in total, but some of them at the beginning are invalid. Currently, checkSignatures fails in this case. A potential issue for this behavior is that a *bad* owner intentionally sends an invalid signature to *veto* the transaction. He can *always* veto if his address is the first (i.e., the smallest) among the owners. On the other hand, a *good* owner is hard to veto some bad transaction if his address is the last (i.e., the largest) among the owners.

No explicit check for the case $2 \leq v \leq 26$ in checkSignatures

According to the signature encoding scheme, a signature with $2 \leq v \leq 26$ is invalid, but the code does not have an explicit check for the case. Instead, it relies on ecrecover to implicitly reject the case. It may be considered to introduce the explicit check for the robustness of the code, as long as the additional gas cost is affordable, since the underlying C implementation of [secp256k1](#) has not been formally verified, and there might exist unknown zero-day vulnerabilities (especially for some corner cases).

handlePayment allows to send Ether to the precompiled contract addresses

handlePayment sends Ether to receiver (in case of `gasToken == address(0)`):

- <https://github.com/gnosis/safe-contracts/blob/v0.1.0/contracts/GnosisSafe.sol#L120>

Here, we see that receiver is non-zero, provided that tx.origin is non-zero. But, receiver could still be a non-owned account, especially one of the precompiled (0x1 - 0x8) contract addresses. Here `receiver.send(amount)` will succeed even with the small gas stipend 2300 for precompiled contracts (at least, for 0x2, 0x3, 0x4, and 0x6). For reference, detailed below is the gas cost for executing each precompiled contract.

Address	Contract	Gas Cost
0x1	ECREC	3,000
0x2	SHA256	$60 + 12 * \text{<byte-size-of-call-data>}$
0x3	RIP160	$600 + 120 * \text{<byte-size-of-call-data>}$
0x4	ID	$15 + 3 * \text{<byte-size-of-call-data>}$
0x5	MODEXP	...
0x6	ECADD	500
0x7	ECMUL	40,000
0x8	ECPAIRING	100,000 + ...

Insufficient external call result check and gas efficiency of transferToken

The transferToken function checks only the termination status (i.e., whether an exception occurred) and the return value of the token contract call to see if the token transfer succeeds. Thus, the GnosisSafe contract may fail the payment if the token contract does not properly implement the ERC20 transfer function. A more obvious way to check the token transfer is to examine the balance of the token-receiver before and after the transfer function call. If the token transfer succeeds, the amount of increase in the balance must be equal to the amount of tokens transferred.

Another concern is about gas efficiency. If the token transfer function returns a large value (or reverts with a large message), it consumes the gas for copying the return value (or the revert message, respectively) to the local memory that is not used at all.

addOwnerWithThreshold in case of contract invariant being unsatisfied

Although it is unlikely, in the case where ownerCount is corrupted (possibly due to the hash collision), ownerCount++ may cause an overflow, resulting in ownerCount being

zero, provided that `threshold == _threshold`. However, in the case where `threshold != _threshold`, if `ownerCount++` contain the overflow, `changeThreshold` will always revert because the following two requirements cannot be satisfied at the same time, where `ownerCount` is zero:

```
// Validate that threshold is smaller than number of owners.  
require(_threshold <= ownerCount, "Threshold cannot exceed owner count");  
// There has to be at least one Safe owner.  
require(_threshold >= 1, "Threshold needs to be greater than 0");
```

signatures byte-size limit

Considering the [current max block gas limit](#) (~8M) and the gas cost for the local memory usage (i.e., $n^2/512 + 3n$ for n bytes), the size of signatures (and other bytes-type arguments) must be (much) less than 2^{16} (i.e., 64KB).

Note that the bytecode generated by the Solidity compiler checks if a bytes-type argument size is less than 2^{32} (bytes), and reverts otherwise.

Common Antipattern Analysis

In this section, we analyze some common antipatterns that have caused failures or losses in past smart contracts. This list includes https://consensys.github.io/smart-contract-best-practices/known_attacks/ as well as <https://blog.sigmaprime.io/solidity-security.html>, and other literature on smart contract security and the experience of our RV team of auditors and formal methods engineers.

1. Re-entrancy vulnerability is present, as described in previous section.
2. Arithmetic over/underflow is possible if the contract invariant is not satisfied, as described in previous section.
3. Unexpected Ether. The default function in Proxy.sol is payable, and Ether is used by GnosisSafe to emit refunds. The contract does not have issues related to presence of a specific amount of Ether.
4. Delegatecall. The payload call performed by GnosisSafe may be not only the regular call, but also a delegatecall or create. The call type is managed by transaction parameter operation, e.g. must be signed by other owners. However, delegatecall is a dangerous type of transaction that can alter the GnosisSafe persistent data in unexpected ways. This danger is properly described in the GnosisSafe documentation. An earlier security audit [for GnosisSafe](#) recommends disabling delegatecall and create entirely unless there is an important use case for it. As it currently stands, it depends on the GnosisSafe client application to properly communicate to the owners the type of call performed, and the dangers involved. This is outside the scope of the present audit.
5. Default Visibilities. All functions have the visibility explicitly declared, and only functions that *must* be public/external are declared as such. Thus no functions use the default public visibility.
6. Entropy Illusion. GnosisSafe does not try to simulate random events. Thus the issue is unrelated to GnosisSafe.
7. Delegating functionality to external contracts. GnosisSafe uses the [proxy pattern](#). Each instantiation of the safe deploys only the lightweight Proxy.sol contract, which delegates (via delegatecall) almost all calls to the proper GnosisSafe.sol deployed in another account. This reduces the cost of instantiating the safe and allows future upgrades. The contract account can upgrade the implementation by calling

GnosisSafe.changeMasterCopy() with the address where the updated GnosisSafe code is deployed. This function can only be called from the proxy account, thus is secure. This pattern presents a security issue when the address of the master cannot be inspected by the contract users, and they have no way to audit its security. In GnosisSafe, master copy can be publicly accessed via Proxy.implementation(), so the issue is not present.

8. Short address/parameter attack. The transaction payload in GnosisSafe is received via transaction parameter data, and then used without changes to initiate an external call. Other external calls are performed using standard methods from Solidity, thus the call data has the correct format. The issue is not present.

9. Unchecked CALL Return Values. Solidity methods call() and send() do not revert when the external call reverts, instead they return false. Some smart contracts naively expect such calls to revert, leading to bugs and potentially security issues. In GnosisSafe, the return value of all such calls is correctly checked.

10. Race Conditions / Front Running. This vulnerability may be present in contracts in which the amount of some Ether/token transfer depends on a sequence of transactions. Thus, an attacker may gain an advantage by manipulating the order of transactions. In GnosisSafe, all the data from which refund token and amount are computed is given as parameters to execTransaction, thus the issue is not present.

11. Denial of Service. Non-owners cannot alter the persistent state of this contract, or use it to call external contracts. Thus no external DoS attack is possible. In principle if an owner loses the private key to his contract and can no longer exercise his duties to sign transactions, this would result in some hindrance. However, the list of owners can always be edited from the contract account, thus it will be a temporary issue.

12. Block Timestamp manipulation. The contract does not use block timestamp.

13. Constructors with Care. Before Solidity v0.4.22, constructor name was the same as the name of the contract. This posed the risk to introduce a dangerous bug if between versions contract would be renamed but constructor would not. GnosisSafe is compiled with Solidity v5.0, where constructors are declared with keyword constructor, thus the issue is not present.

14. Uninitialised local storage variables. Not used in GnosisSafe.

15. Floating Points and Numerical Precision. Floating point numbers are not used in GnosisSafe.



16. Tx.Origin Authentication. In GnosisSafe tx.origin is not used for authentication.

17. Constantinople gas issue. The issue may appear only in contracts without explicit protection for re-entrancy. We already discussed re-entrancy on point 1.

Formal Specification & Verification Overview

Here we provide the background and overview of the formal specification and verification artifact of GnosisSafe.

Formal Verification Methodology

Our methodology for formal verification of smart contracts is as follows. First, we formalize the high-level business logic of the smart contracts, based on a typically informal specification provided by the client, to provide us with a precise and comprehensive specification of the functional correctness properties of the smart contracts. This high-level specification needs to be confirmed by the client, possibly after several rounds of discussions and changes, to ensure that it correctly captures the intended behavior of their contracts. Then we refine the specification all the way down to the Ethereum Virtual Machine (EVM) level, often in multiple steps, to capture the EVM-specific details. The role of the final EVM-level specification is to ensure that nothing unexpected happens at the bytecode level, that is, that only what was specified in the high-level specification will happen when the bytecode is executed. To precisely reason about the EVM bytecode without missing any EVM quirks, we adopted [KEVM](#), a complete formal semantics of the EVM, and instantiated the [K-framework reachability logic theorem prover](#) to generate a correct-by-construction deductive program verifier for the EVM. We use the verifier to verify the compiled EVM bytecode of the smart contract against its EVM-level specification. Note that the Solidity compiler is not part of our trust base, since we directly verify the compiled EVM bytecode. Therefore, our verification result does not depend on the correctness of the Solidity compiler.

For more details, resources, and examples, we refer the reader to our Github repository for formal verification of smart contracts, publicly available at:

<https://github.com/runtimeverification/verified-smart-contracts>

Resources

We use the [K-framework](#) and its verification infrastructure throughout the formal verification effort. All of the formal specifications are mechanized within the K-framework as well. Therefore, some background knowledge about the K-framework would be necessary for reading and fully understanding the formal specifications and reproducing

the mechanized proofs. We refer the reader to the following resources for background knowledge about the K-framework and its verification infrastructure.

- [K-framework](#)
 - [Download and install](#)
 - [K tutorial](#)
 - [K editor support](#)
- [KEVM](#): an executable formal semantics of the EVM in K
 - [Jellopaper](#): reader-friendly formatting of KEVM
 - [KEVM technical report](#)
- [K reachability logic prover](#)
 - [eDSL](#): domain-specific language for EVM-level specifications
- [ERC20-K](#): a formal specification of the high-level business logic of [ERC20](#)
- [ERC20-EVM](#): an EVM-level refinement of ERC20-K
- [ERC777-K](#): a formal specification of the high-level business logic of [ERC777](#)

Mechanized Specification and Proof

Following our formal verification methodology described above, we formalized the high-level specification of the GnosisSafe contract, and refined the specification all the way down to the Ethereum Virtual Machine (EVM) level to capture the EVM-specific details.

The fully mechanized, EVM-level formal specification that we verified against the GnosisSafe contract *bytecode*, the code released with [version 0.1.0](#) (commit ID [427d6f7](#)) on the `gnosis/safe-contracts` Github repository, is available at:

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini>

Note that our verification result is valid only for the aforementioned bytecode. Any change to the bytecode may invalidate all our claims, findings, and recommendations.

The formal specification is fully mechanized within and automatically verified by our EVM verifier, a correct-by-construction deductive program verifier derived from [KEVM](#) and [K-framework](#)'s [reachability logic theorem prover](#).

Below are some statistics of the mechanized formal specification:

- Size of the mechanized formal specification: ~2,000 LOC

- Number of properties (called *reachability claims*) in the specification: 65
- Total verification time: 29,103s (~8h) @ Intel i7-4960X CPU 3.60GHz
- Average number of symbolic execution steps with the KEVM semantic rules taken to verify each reachability claim: 5,050 (max: 11,635)

The specification is written in [eDSL](#), a domain-specific language for EVM specifications, which the reader must understand in order to thoroughly comprehend our EVM-level specifications. Refer to [resources](#) for background on our technology. The full K reachability logic specifications are automatically derived from the provided [eDSL](#) specification.

Run the following command in the root directory of [the verified-smart-contracts Github repository](#), and it will generate the full specifications under the directory `specs/gnosis`:

```
$ make -C gnosis all
```

Run the EVM verifier to prove that the specifications are satisfied by (the compiled EVM bytecode of) the target functions. See these [instructions](#) for more details of running the verifier.

Formal Specification Details

Now we describe the details of the formal specification that we verified against the GnosisSafe contract. We first clarify our assumption (i.e., what is *not* verified), and then describe the formal specification for each function we verified.

Assumptions

We found that certain input states (including function argument values and unknown external accounts' state) may lead to the failure of the contract satisfying the desired properties, although some of those failure cases are not likely to happen in practice. For the failure cases that are possible to happen, we carefully reviewed and provided the details of our analysis and suggestions in the previous section (see the [List of Findings](#) section).

In order to verify that the contract satisfies the desired properties *except for those failure cases*, we had to assume that the input states are adequate (i.e., assuming the negation of the failure conditions). Below we compiled a list of the assumptions (i.e., pre-conditions) we made. Some of those assumptions are general, while others are specific to certain functions. The function-specific assumptions will be clarified in subsequent sections, where we describe the formal specification of each function as we formally verify it.

We note that it is the sole responsibility of the developers of the contract (and their clients, respectively) to ensure that the assumptions are met whenever they update (and use, respectively) the contract.

No wrap-around overflow:

- threshold is assumed small enough to avoid overflow (wrap-around).
- nonce is assumed small enough to avoid overflow (wrap-around).

If an overflow happens and the value is wrapped around, the contract will be in an unexpected state, and may not work properly thereafter. However, we note that the overflow case is not likely to happen, considering the resource limitation (such as gas).

Well-formed input:

- The address-type argument (and storage) values are within the range of address, i.e., $[0, 2^{160}-1]$, inclusive. Otherwise, the first 96 ($= 256 - 160$) bits are silently truncated (with no exception).
- No overlap between multiple memory chunks of byte-typed arguments. Otherwise, the function becomes nondeterministic.
- (Only for signatureSplit) No list index out of bounds.
- (Only for checkSignatures) Every signature encoding is well-formed. Otherwise, the function becomes nondeterministic.

If the input well-formedness conditions are not met, the function may not work as expected, and its behavior depends on the VM state when the function is called.

We note that these conditions are satisfied for all internal functions in the current GnosisSafe contract. For the external functions, however, it is the responsibility of any client of this contract to ensure that these conditions are met when they prepare for the function call data.

Non-interfering external contract call:

- The external contract call does not change the current (i.e., the proxy) storage.

Roughly speaking, the non-interfering condition rules out the possibility of reentrancy. In other words, this assumption requires any client of the contract to ensure that they do not send a user transaction to an external contract without knowing what the external contract does.

Trusted ERC20 token contract:

- The gasToken contract properly implements the ERC20 transfer function.

In case of token payment, the given token contract is called for transferring tokens. However, the GnosisSafe contract checks only the termination status (i.e., whether an exception occurred) and the return value of the token contract call to see if the token transfer succeeds. Thus, if the token contract does not implement the transfer function properly, the GnosisSafe contract may fail the payment. It is the responsibility of any client of this contract to ensure that a valid ERC20 token contract is provided for the token payment.

GnosisSafe contract

Function signatureSplit

`signatureSplit` is an internal function that takes a sequence of signatures and an index, and returns the indexed signature as a tuple of its `v`, `r`, and `s` fields.

```
function signatureSplit(bytes memory signatures, uint256 pos)
    internal
    pure
    returns (uint8 v, bytes32 r, bytes32 s)
```

Stack and memory:

The function takes two inputs, signatures and pos, where signatures is passed through the memory while pos is through the stack.

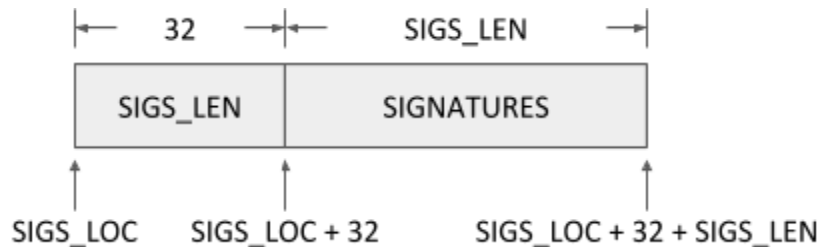
The input stack is given as follows:

POS : SIGS_LOC : RETURN_LOC : WS

where POS is the value of pos, and SIGS_LOC is the starting location of the memory that stores the signatures byte buffer.

NOTE: Throughout this specification, RETURN_LOC is the return address (PC value), and WS is the caller's stack frame, which are not relevant for the current function's behavior.

The memory stores the signatures buffer starting at the location SIGS_LOC, where it first stores the size of the buffer SIGS_LEN, followed by the actual buffer SIGNATURES, as illustrated below:



The function's return value is a tuple of (v, r, s) , which is pushed into the stack, as in the following output stack:

RETURN_LOC : S : R : V : WS

where

- R: 32 bytes from the offset $65 * \text{POS}$ of SIGNATURES
- S: 32 bytes from the offset $65 * \text{POS} + 32$ of SIGNATURES
- V: 1 byte at the offset $65 * \text{POS} + 64$ of SIGNATURES

Function visibility and modifiers:

The function cannot be directly called from outside, as it is internal. An external call to this function will silently terminate with no effect (and no exception).

The function does not update the storage, as it is marked pure.

Exceptions:

If one of the following no-overflow conditions is *not* met, the function will throw or revert:

- The input stack size should be small enough to avoid the stack overflow.
- The maximum memory location accessed, i.e., $\text{SIGS_LOC} + 32 + (65 * \text{POS} + 65)$, should be small enough to avoid the integer overflow for the pointer arithmetic.

Pre-conditions:

Well-formed input:

- No index out of bounds, i.e., $(\text{POS} + 1) * 65 \leq \text{SIGS_LEN}$

We note that the input well-formedness condition is satisfied for all internal uses of this function in the current GnosisSafe contract.

Mechanized formal specification:

Below is the specification that we verified against the GnosisSafe contract bytecode.

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L54-L89>

Function encodeTransactionData

`encodeTransactionData` is a public function that calculates the hash value of the given transaction data.

```
function encodeTransactionData(  
    address to,  
    uint256 value,  
    bytes memory data,  
    Enum.Operation operation,  
    uint256 safeTxGas,  
    uint256 dataGas,  
    uint256 gasPrice,  
    address gasToken,  
    address refundReceiver,  
    uint256 _nonce  
)  
  
    public  
    view  
    returns (bytes memory)
```

Stack and memory:

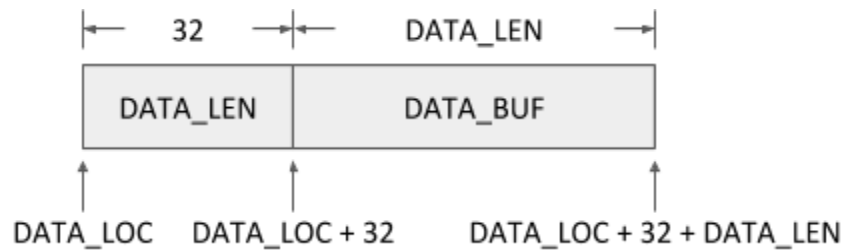
The function is public, to which both internal and external calls can be made. One of the main differences between the two types of calls is how to pass the input. The internal call passes the input through the stack and the memory, while the external call passes the input through the call data.

For the internal call, the input stack is given as follows:

```
NONCE : REFUND_RECEIVER : GAS_TOKEN : GAS_PRICE : DATA_GAS : SAFE_TX_GAS :  
OPERATION : DATA_LOC : VALUE : TO : RETURN_LOC : WS
```

where the first ten elements are the function arguments in reverse order, while `DATA_LOC` is a memory pointer to the actual buffer of data. Note that `OPERATION` is encoded as `unit8`.

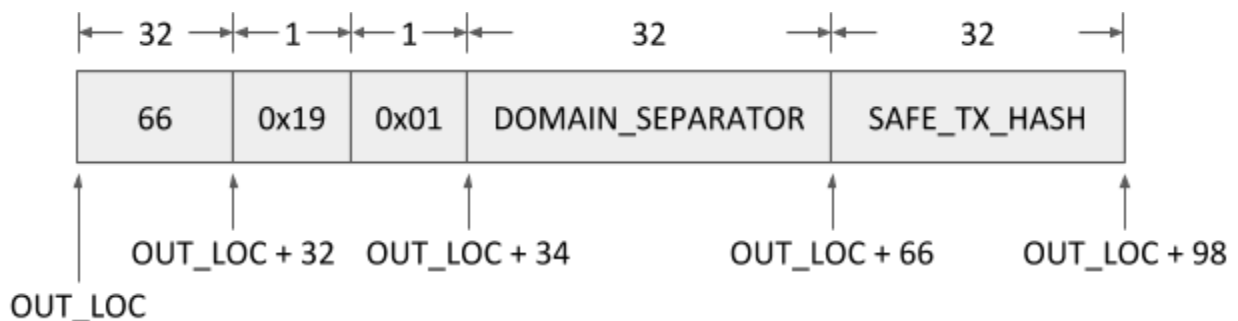
The memory stores the data buffer starting at the location DATA_LOC, where it first stores the size of the buffer, followed by the actual buffer bytes, as illustrated below:



The output stack consists of:

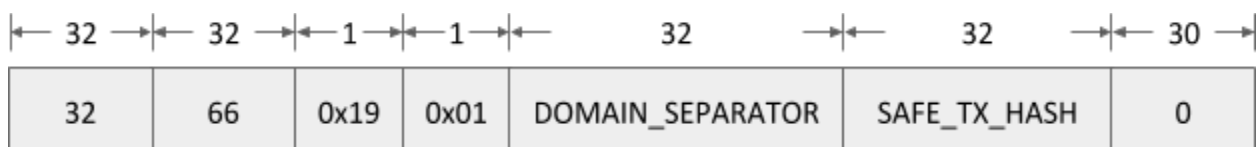
RETURN_LOC : OUT_LOC : WS

For the internal call, the return value (buffer) is passed through the memory, being stored at the starting location OUT_LOC, as follows:



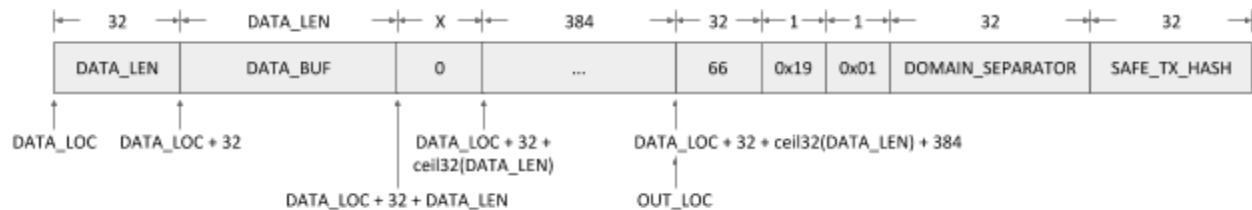
Here the first 32 bytes denote the size of the buffer, and the remaining 66 bytes denote the result of `abi.encodePacked(byte(0x19), byte(0x01), domainSeparator, safeTxHash)`. Note that the first two elements, 0x19 and 0x01, are not aligned, because of the use of `abi.encodePacked` instead of `abi.encode`. Also, SAFE_TX_HASH is the result of `abi.encode(SAFE_TX_TYPEHASH, to, value, keccak256(data), operation, safeTxGas, dataGas, gasPrice, gasToken, refundReceiver, _nonce)`, where each argument is 32-byte aligned with zero padding on the left.

For the external call, on the other hand, the return value (buffer) is encoded, in the ABI format, as follows:



Here the prefix (the first 32 bytes) and the postfix (the last 30 bytes) are attached, compared to that of the internal call. The prefix is the offset to the start of the return value buffer, and the postfix is the zero padding for the alignment.

For the internal call, the output memory is as follows:



where $X = \text{ceil32}(\text{DATA_LEN}) - \text{DATA_LEN}$. Here the function writes to the memory starting from **DATA_LOC + 32 + ceil32(DATA_LEN)**. The first 384 bytes are used for executing keccak256 to compute safeTxHash, i.e., 352 bytes for preparing for 11 arguments ($= 32 * 11$), and 32 bytes for holding the return value. The next 98 bytes are used for passing the return value, as described above.

Note that the external call results in the same output memory, but the memory is not shared by the caller, and does not affect the caller's memory.

Function visibility and modifiers:

The function does not update the storage, as it is marked view.

For the external call, msg.value must be zero, since the function is not payable. Otherwise, it throws.

Exceptions:

If one of the following no-overflow conditions is *not* met, the function will throw or revert:

- For the external call, the call depth should be small enough to avoid the call depth overflow.
- For the internal call, the input stack size should be small enough to avoid the stack overflow.
- The maximum memory location accessed, i.e., $\text{DATA_LOC} + 32 + \text{ceil32}(\text{DATA_LEN}) + 482$, should be small enough to avoid the integer overflow for the pointer arithmetic.

If one of the following input well-formedness conditions is *not* met, the function will throw or revert:

- The operation should be either 0, 1, or 2. Otherwise, the execute function (defined at Executor.sol) will throw.
- The byte size of data should be less than 2^{32} . Otherwise, it reverts.

Pre-conditions:

Well-formed input:

- The to, gasToken, and refundReceiver argument values are all within the range of address, i.e., $[0, 2^{160}-1]$, inclusive. Otherwise, the first 96 (= 256 - 160) bits are silently truncated (with no exception).

We note that the input well-formedness condition is satisfied for all internal uses of this function in the current GnosisSafe contract.

Mechanized formal specification:

Below are the specifications that we verified against the GnosisSafe contract bytecode.

For internal call:

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L98-L223>

For external call:

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L252-L328>

Function `handlePayment`

`handlePayment` is a private function that pays the gas cost to the receiver in either Ether or tokens.

```
function handlePayment(  
    uint256 startGas,  
    uint256 dataGas,  
    uint256 gasPrice,  
    address gasToken,  
    address payable refundReceiver  
)  
  
private
```

Stack and memory:

All of the input arguments are passed through the stack, and no memory is required since they are all fixed-size:

REFUND_RECEIVER : GAS_TOKEN : GAS_PRICE : DATA_GAS : START_GAS : RETURN_LOC : WS

The function has no return value, and thus the output stack, if succeeds, is as follows:

RETURN_LOC : WS

State update:

The payment amount is calculated by the following formula:

$$((\text{START_GAS} - \text{GAS_LEFT}) + \text{DATA_GAS}) * \text{GAS_PRICE}$$

where `GAS_LEFT` is the result of `gasleft()` at [line 115](#).

If an arithmetic overflow occurs when evaluating the above formula, the function reverts.

If no overflow occurs, receiver is set to `tx.origin` if `refundReceiver` is zero, otherwise it is set to `refundReceiver`. Thus receiver is non-zero.

Finally, the amount of Ether or tokens is sent to receiver. If the payment succeeds, the function returns (with no return value). Otherwise, it reverts. There are two payment methods, and each method has the following success/failure behaviors:

- Ether payment:
 - If send succeeds, then the function returns (with no return value).
 - Otherwise, it reverts.
- Token payment:
 - If `gasToken.transfer()` succeeds (i.e., no exception):
 - If `gasToken.transfer()` returns nothing, the function returns.
 - If `gasToken.transfer()` returns a (32-byte) non-zero value, it returns.
 - If `gasToken.transfer()` returns zero, it reverts.
 - Otherwise, it reverts.
 - If `gasToken.transfer()` throws or reverts, the function reverts regardless of the return value of `gasToken.transfer()`.

Here, we have little concern about the reentrancy for `send` or `gasToken.transfer()`, since there is no critical statement after `send`/`transfer`, and also the function is private.

Function visibility and modifiers:

The function cannot be directly called from outside, as it is private. An external call to this function will silently terminate with no effect (and no exception).

Exceptions:

If one of the following no-overflow conditions is *not* met, the function will throw or revert:

- The input stack size should be small enough to avoid the stack overflow.

Pre-conditions:

Well-formed input:

- The value of the address arguments are within the range of address, i.e., $[0, 2^{160}-1]$, inclusive. Otherwise, the first 96 (= 256 - 160) bits are silently truncated (with no exception).

We note that the input well-formedness condition is satisfied for all internal uses of this function in the current GnosisSafe contract.



Trusted ERC20 token contract:

- The gasToken contract properly implements the ERC20 transfer function.

Mechanized formal specification:

Below is the specification that we verified against the GnosisSafe contract bytecode.

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L438-L563>

Function checkSignatures

`checkSignatures` is an internal function that checks the validity of the given signatures.

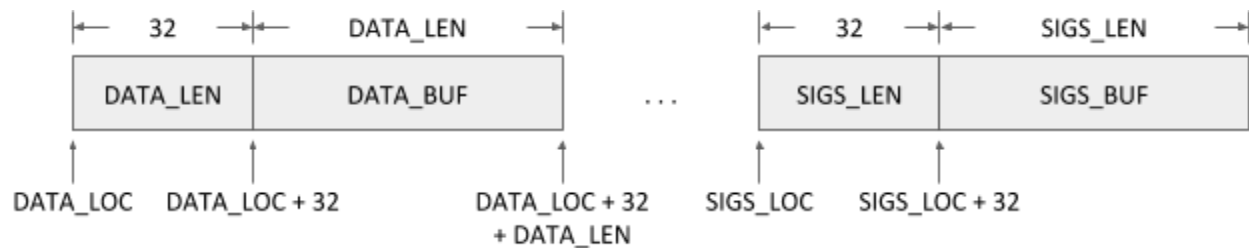
```
function checkSignatures(  
  bytes32 dataHash,  
  bytes memory data,  
  bytes memory signatures,  
  bool consumeHash  
)  
  
  internal  
  returns (bool)
```

Stack and memory:

The input arguments are passed through the stack as follows:

CONSUME_HASH : SIGS_LOC : DATA_LOC : DATA_HASH : RETURN_LOC : WS

where data and signatures are stored in the memory:



The function returns true if:

- the number of signatures is more than equal to threshold, and
- the first threshold number of signatures are valid, signed by owners, and sorted by their owner address.

where a signature is valid if:

- case `v = 0`: `r`'s `isValidSignature` returns true.
- case `v = 1`: `r == msg.sender` or `dataHash` is already approved.

- otherwise: it is a valid ECDSA signature.

Otherwise, the function returns false, unless isValidSignature throws (or reverts).

If isValidSignature throws or reverts, checkSignatures reverts, immediately terminating without returning to execTransaction.

Also, if consumeHash = true, the function may update approvedHashes[currentOwner][dataHash] to zero.

Function visibility and modifiers:

The function cannot be directly called from outside, as it is internal. An external call to this function will silently terminate with no effect (and no exception).

Exceptions:

If one of the following no-overflow conditions is *not* met, the function will throw or revert:

- The input stack size should be small enough to avoid the stack overflow.
- The maximum memory location accessed should be small enough to avoid the integer overflow for the pointer arithmetic.

If one of the following input well-formedness conditions is *not* met, the function will throw or revert:

- The byte size of data should be less than 2^{32} . Otherwise, it reverts.

Pre-conditions:

No wrap-around:

- threshold is small enough to avoid overflow (wrap-around).

Well-formed input:

- Every owner (i.e., some o such that $owners[o] \neq 0$) is within the range of address. Otherwise, the function simply truncates the higher bits when validating the signatures.
- No overlap between two memory chunks of data and signatures, i.e., $DATA_LOC + 32 + DATA_LEN \leq SIGS_LOC$. Otherwise, the function becomes nondeterministic.

- Every signature encoding is well-formed. Otherwise, the function becomes nondeterministic.

We note that the first two input well-formedness conditions are satisfied for all internal uses of this function in the current GnosisSafe contract. However, the last condition should be satisfied by the client when he calls `execTransaction`, since the current contract omits the well-formedness check of the signature encoding.

Non-interfering external contract call:

- The external contract call does not change the current (i.e., the proxy) storage.

Formal specification (at a high-level):

We formalize the validity of (arbitrary number of) signatures in a way that we can avoid explicit quantifier reasoning during the mechanized formal verification, as follows.

We first define the `the-first-invalid-signature-index` as follows: (The mechanized definition is [here](#).)

- A1: For all $i < \text{the-first-invalid-signature-index}$, `signatures[i]` is valid.
- A2: `signatures[the-first-invalid-signature-index]` is NOT valid.

Now we can formulate the behavior of `checkSignatures` using the above definition (with no quantifiers!) as follows:

- T1: `checkSignatures` returns true if `the-first-invalid-signature-index` \geq threshold.
- T2: Otherwise, returns false.

To prove the above top-level specifications, T1 and T2, we need the following loop invariant:

For some i such that $0 \leq i < \text{threshold}$ and $i \leq \text{the-first-invalid-signature-index}$:

- L1: If $i < \text{threshold} \leq \text{the-first-invalid-signature-index}$, then the function returns true once the loop terminates.
- L2: Else (i.e., if $i \leq \text{the-first-invalid-signature-index} < \text{threshold}$), then the function eventually returns false.

To prove the above loop invariant, L1 and L2, we need the following claims for a single loop iteration:

- M1: If signatures[i] is valid, it continues to the next iteration (i.e., goes back to the loop head).
- M2: If signatures[i] is NOT valid, it returns false.

Proof sketch:

The top level specification:

- T1: By L1 with $i = 0$.
- T2: By L2 with $i = 0$.

The loop invariant:

- L1: By A1, signatures[i] is valid. Then by M1, it goes back to the loop head, and we have two cases:
 - Case 1: $i + 1 = \text{threshold}$: It jumps out of the loop, and return true.
 - Case 2: $i + 1 < \text{threshold}$: By the circular reasoning with L1.
- L2:
 - Case 1: $i = \text{the-first-invalid-signature-index}$: By A2, signatures[i] is NOT valid. Then, by M2, we conclude.
 - Case 2: $i < \text{the-first-invalid-signature-index}$: By A1, signatures[i] is valid. Then, by M1, it goes to the loop head, and by the circular reasoning with L2, we conclude (since we know that $i + 1 \leq \text{the-first-invalid-signature-index} < \text{threshold}$).

The single loop iteration claim does not involve the recursive structure, and thus can be verified in the similar way as other specifications.

Mechanized formal specification:

Below is the specification that we verified against the GnosisSafe contract bytecode.

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L695-L1155>

Function execTransaction

`execTransaction` is an external function that executes the given transaction.

```
function execTransaction(  
    address to,  
    uint256 value,  
    bytes calldata data,  
    Enum.Operation operation,  
    uint256 safeTxGas,  
    uint256 dataGas,  
    uint256 gasPrice,  
    address gasToken,  
    address payable refundReceiver,  
    bytes calldata signatures  
)  
  
external  
returns (bool success)
```

We consider only the case of `Enum.Operation.Call` operation (i.e., `operation == 0`). The other two cases are out of the scope of the current engagement.

Stack and memory:

Since it is an external function, it starts with a fresh VM (i.e., both the stack and the memory are empty, the PC is 0, etc.)

State update:

The function checks the validity of signatures, and reverts if not valid.

Then it increases nonce, and calls `execute` with the given transaction.

It finally calls `handlePayment`.

The function has the following non-trivial behaviors:

- `checkSignatures` may revert, which immediately terminates the current VM, without returning to `execTransaction`.

- execute does NOT revert, even if the given transaction execution throws or reverts. The return value of the given transaction, if any, is silently ignored.
 - However, execute may still throw for some cases (e.g., when operation is not within the range of Enum.Operation).
- handlePayment may throw or revert, and in that case, execTransaction reverts (i.e., the given transaction execution is reverted as well, and no ExecutionFailed event is logged).

Function visibility and modifiers:

msg.value must be zero, since the function is not payable. Otherwise, it throws.

Exceptions:

If one of the following input well-formedness conditions is *not* met, the function will throw or revert:

- The byte size of data and signatures should be less than 2^{32} . Otherwise, it reverts.

Pre-conditions:

No wrap-around:

- nonce is small enough to avoid overflow (wrap-around).

Well-formed input:

- The value of the address arguments are within the range of address, i.e., $[0, 2^{160}-1]$, inclusive.

Non-interfering external contract call:

- The external contract call does not change the current (i.e., the proxy) storage.

Mechanized formal specification:

Below is the specification that we verified against the GnosisSafe contract bytecode.

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L1157-L1406>

OwnerManager contract

The OwnerManager contract maintains the set of owners.

The storage state of owners represents a (non-empty) list of (o_0, o_1, \dots, o_N) , which denotes the (possibly empty) set of owners $\{o_1, \dots, o_N\}$. (Note that o_0 is a dummy element of the list, not an owner.)

The OwnerManager contract must satisfy the following contract invariant, once initialized (after setup):

- $\text{ownerCount} \geq \text{threshold} \geq 1$
- ownerCount is small enough to avoid overflow
- owners represents the list of (o_0, o_1, \dots, o_N) such that:
 - $N = \text{ownerCount}$
 - o_i is non-zero (for all $0 \leq i \leq N$)
 - $o_0 = 1$
 - all o_i 's are distinct (for $0 \leq i \leq N$)
 - $\text{owners}[o_i] = o_{\{i+1 \bmod N+1\}}$ for $0 \leq i \leq N$
 - $\text{owners}[x] = 0$ for any x not in the list (o_0, \dots, o_N)

Function addOwnerWithThreshold

`addOwnerWithThreshold` is a public authorized function that adds a new owner and updates threshold.

```
function addOwnerWithThreshold(address owner, uint256 _threshold)
    public
    authorized
```

State update:

Suppose `owners` represents (o_0, o_1, \dots, o_N) and the contract invariant holds before calling the function. Note that the contract invariant implies $N \geq 1$.

The function reverts if one of the following input conditions is not satisfied:

- The argument `owner` should be a non-zero new owner, i.e., `owner != 0` and `owner != o_i` for all $0 \leq i \leq N$.
- The argument `_threshold` should be within the range of $[1, N+1]$, inclusive.

NOTE: The check `require(owner != SENTINEL_OWNERS)` is logically redundant in the presence of `require(owners[owner] == address(0))` and the given contract invariant.

If the function succeeds, the post state will be:

- `owners` will represent $(o_0, owner, o_1, \dots, o_N)$.
- `ownerCount` = $N+1$
- `threshold` = `_threshold`

Function visibility and modifiers:

The function should be invoked by the proxy account. Otherwise, it reverts.

Mechanized formal specification:

Below is the specification that we verified against the GnosisSafe contract bytecode.

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L1412-L1581>

Function removeOwner

`removeOwner` is a public authorized function that removes the given owner and updates threshold.

```
function removeOwner(address prevOwner, address owner, uint256 _threshold)
    public
    authorized
```

State update:

Suppose `owners` represents (o_0, o_1, \dots, o_N) and the contract invariant holds before calling the function. Note that the contract invariant implies $N \geq 1$.

The function reverts if one of the following input conditions is not satisfied:

- $N \geq 2$
- There exists $0 \leq k < N$ such that `prevOwner` = `ok` and `owner` = `o{k+1}`.
- The argument `_threshold` should be within the range of $[1, N-1]$, inclusive.

NOTE: The check `require(owner != SENTINEL_OWNERS)` is necessary to ensure $k \neq N$.

If the function succeeds, the post state will be:

- `owners` will represent $(\dots, o_k, o_{k+2}, \dots)$ for $0 \leq k < N$.
- `ownerCount` = $N-1$
- `threshold` = `_threshold`

Function visibility and modifiers:

The function should be invoked by the proxy account. Otherwise, it reverts.

Mechanized formal specification:

Below is the specification that we verified against the GnosisSafe contract bytecode.

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L1583-L1716>

Function swapOwner

`swapOwner` is a public authorized function that replaces `oldOwner` with `newOwner`.

```
function swapOwner(address prevOwner, address oldOwner, address newOwner)
    public
    authorized
```

State update:

Suppose `owners` represents (o_0, o_1, \dots, o_N) and the contract invariant holds before calling the function. Note that the contract invariant implies $N \geq 1$.

The function reverts if one of the following input conditions is not satisfied:

- The argument `newOwner` should be a non-zero new owner, i.e., `newOwner != 0` and `newOwner != o_i` for all $0 \leq i \leq N$.
- There exists $0 \leq k < N$ such that `prevOwner = o_k` and `oldOwner = o_{k+1}`.

NOTE:

- The check `require(newOwner != SENTINEL_OWNERS)` is logically redundant in the presence of `require(owners[newOwner] == address(0))` and the given contract invariant.
- The check `require(oldOwner != SENTINEL_OWNERS)`, however, is necessary to ensure $k \neq N$.

If the function succeeds, the post state will be:

- `owners` will represent $(\dots, o_k, \text{newOwner}, \dots)$ for $0 \leq k < N$.
- `ownerCount` and `threshold` are not updated.

Function visibility and modifiers:

The function should be invoked by the proxy account. Otherwise, it reverts.

Mechanized formal specification:

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L1718-L1805>

ModuleManager contract

The ModuleManager contract maintains the set of modules.

The storage state of modules represents a (non-empty) list of (m_0, m_1, \dots, m_N) , which denotes the (possibly empty) set of modules $\{m_1, \dots, m_N\}$. (Note that m_0 is a dummy element of the list, not a module.)

The ModuleManager contract must satisfy the following contract invariant, once initialized (after setup):

- modules represents the list of (m_0, m_1, \dots, m_N) such that:
 - $N \geq 0$
 - m_i is non-zero (for all $0 \leq i \leq N$)
 - $m_0 = 1$
 - all m_i 's are distinct (for $0 \leq i \leq N$)
 - $\text{modules}[m_i] = m_{\{i+1 \bmod N+1\}}$ for $0 \leq i \leq N$
 - $\text{modules}[x] = 0$ for any x not in the list (m_0, \dots, m_N)

Note that the set of modules could be empty, while the set of owners cannot.

Function enableModule

`enableModule` is a public authorized function that adds a new module.

```
function enableModule(Module module)
    public
    authorized
```

State update:

Suppose `modules` represents (m_0, m_1, \dots, m_N) and the contract invariant holds before calling the function.

The function reverts if one of the following input conditions is not satisfied:

- The argument module should be a non-zero new module, i.e., $module \neq 0$ and $module \neq m_i$ for all $0 \leq i \leq N$.

NOTE: The check `require(module != SENTINEL_OWNERS)` is logically redundant in the presence of `require(modules[address(module)] == address(0))` and the given contract invariant.

If the function succeeds, the post state will be:

- `modules` will represent $(m_0, module, m_1, \dots, m_N)$.

Function visibility and modifiers:

The function should be invoked by the proxy account. Otherwise, it reverts.

Mechanized formal specification:

Below is the specification that we verified against the GnosisSafe contract bytecode.

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L1811-L1874>

Function disableModule

`disableModule` is a public authorized function that removes the given module.

```
function disableModule(Module prevModule, Module module)
    public
    authorized
```

State update:

Suppose `modules` represents (m_0, m_1, \dots, m_N) and the contract invariant holds before calling the function.

The function reverts if one of the following input conditions is not satisfied:

- $N \geq 1$
- There exists $0 \leq k < N$ such that `prevModule = m_k` and `module = m_{k+1}`.

NOTE: The check `require(module != SENTINEL_OWNERS)` is necessary to ensure $k \neq N$ and $N \geq 1$.

If the function succeeds, the post state will be:

- `modules` will represent $(\dots, m_k, m_{k+2}, \dots)$ for $0 \leq k < N$.

Function visibility and modifiers:

The function should be invoked by the proxy account. Otherwise, it reverts.

Mechanized formal specification:

Below is the specification that we verified against the GnosisSafe contract bytecode.

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L1876-L1939>

Function `execTransactionFromModule`

`execTransactionFromModule` is a public function that executes the given transaction.

```
function execTransactionFromModule(  
    address to, uint256 value, bytes memory data, Enum.Operation operation  
)  
  
    public  
  
    returns (bool success)
```

Here we consider only the case that `modules` denotes the empty set. The case for a non-empty set of modules is out of the scope of the current engagement.

The function reverts if `msg.sender != 1` and `modules` denotes the empty set, i.e., `modules[x] = 0` for any `x != 1`, and `modules[1] = 1`.

Mechanized formal specification:

Below is the specification that we verified against the GnosisSafe contract bytecode.

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L1941-L1981>

MasterCopy contract

Function changeMasterCopy

`changeMasterCopy` is a public authorized function that updates `masterCopy`.

```
function changeMasterCopy(address _masterCopy)
    public
    authorized
```

State update:

The function reverts if the argument `_masterCopy` is zero.

Otherwise, it updates `masterCopy` to `_masterCopy`.

Function visibility and modifiers:

The function should be invoked by the proxy account. Otherwise, it reverts.

Mechanized formal specification:

Below is the specification that we verified against the GnosisSafe contract bytecode.

<https://github.com/runtimeverification/verified-smart-contracts/blob/ee8e6c8763dfa57d0372a3a67ed4df2c54fcea5e/gnosis/gnosis-spec.ini#L1987-L2037>