Security Audit Report

GnosisSafe Contract

Delivered: February 7th, 2019

Updated: February 27th, 2019

Prepared for Gnosis Ltd. by

runtime verification

Table of Contents

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 Solidity contract of GnosisSafe as part of the formal verification engagement of the said contract. The audit focuses on security-critical properties that were identified by the Gnosis team, and we manually reason about these properties by examining both the Solidity source code and its compiled EVM bytecode.

The audit 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.

This audit report is part of the full formal verification report.

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 Solidity contract of GnosisSafe against the security-critical properties identified by the Gnosis team. The contract source code is from commit ID 427d6f7e779431333c54bcb4d4cde31e4d57ce96 of the gnosis/safe-contracts Github repository.

The scope of the audit 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 audit 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.

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 vectors[1], 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 <= v <= 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 ISingatureValidator. 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/0x0000000000000000000000000000000000000000) 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. The 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, 2^160-1] inclusive. Otherwise, the fist 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 <= v <= 26 in checkSignatures

According to the signature encoding scheme, a signature with 2 <= v <= 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 * <byte-size-of-call-data>
0x3	RIP160	600 + 120 * <byte-size-of-call-data>
0x4	ID	15 + 3 * <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.