Gnosis Safe Audit Report

This report contains 2 parts, an interim report, and “iteration 2”.

Interim Report on the Audit of Gnosis Safe smart contracts

Scope

The code in the scope of this audit can be access using this link:

https://github.com/gnosis/gnosis-safe-contracts/tree/b1a80f23b99c4983229dbb678c9f77895c9d2b70

File contracts/libraries/MultiSendStruct.sol is out of scope, because it was just a test of experimental Solidity support for ABI version 2.

Method

Two methods have been used to audit the code so far: code inspection (i.e. code review), and symbolic execution in a spreadsheet. Symbolic execution can also be thought as a byte code inspection, because it highlights the details of the byte code emitted by the current version of Solidity (version 0.4.20) when compiling the smart contracts. Additional methods (e.g. some primitive formal verification techniques) might be used for the final report.

Overview of the contracts

Gnosis Safe is a complete rewrite of widely used and so far secure Gnosis Multisig wallet. It contains improvements like (list not extensive):

- better compatibility with some token contracts, thanks to using low-level assembly for calling other contracts
- more modular design thanks to extensions
- more efficient (in code size) deployment and upgradeability, thanks to functional proxies made possible after Ethereum’s Byzantium fork

GnosisSafe.sol

Function *executeExtension* allows invocations using **Call**, **Create**, and **DelegateCall** after an extension has been whitelisted (though all three example extensions in the code base assert that only **Call** is used). **DelegateCall** is inherently much more dangerous, because it can potentially overwrite any storage of the Gnosis Safe caller. Recommendation is to only support Call, unless there is an important use case for Create and DelegateCall. If in the future versions Create or DelegateCall are required, they can be added via self-upgrading mechanism.

Field *masterCopy* is supposed to be at the same storage index as the corresponding masterCopy field in the Proxy.sol. It is not currently enforced by the compiler. Recommendation - use comments in both GnosisSafe.sol and Proxy.sol that masterCopy fields are related to each other and to MUST stay in the same storage index for the upgradeability to work, and for the safety of using GnosisSafe via Proxy.

During the byte code inspection, it has been noticed that the field *threshold* (of type uint8) is sharing the 32-byte word with the *masterCopy* field. Thanks to all the precautions that Solidity compiler is taking to separate the two fields during all accesses, it does not pose a problem

currently. However, for the future robustness, the Recommendation is to either document this in comments, or make *threshold* a usual *int* (256 bit) field, so it does not share 32-byte with *masterCopy* anymore.

In function *confirmTransaction*, *_nonce* argument is not used, needs to be passed into the *getTransactionHash* call. Recommendation: this is apparently a typo, fix it.

Inside the loop of setup method, every occurrence **_owners[i]** generates the code for loading from memory to stack. Recommendation: assign _owners[i] to an intermediate variable - it will result in smaller code, will consume less gas, and will be easier to audit.

Since *setup* function is called from the constructor, and can also be used standalone, the compiler actually generates two copies of this function. These copies are almost the same, the difference is in the way the input argument are read. When *setup* is called from the constructor, the input arguments are appended at the very end of the transaction data and are access by using CODECOPY. When setup is called standalone, the input arguments are accessed via CALLDATALOAD or CALLDATACOPY. Recommendation - remove call to *setup* from the GnosisSafe constructor. The master copy setup will require two transactions instead of one, but the benefit is shorter code and easier audit.

Function *executeTransaction* embodies the main complexity of this smart contract, and therefore needs to be as simple as possible. Recommendation: **_owners[j]** is used three times inside the loop, the use of intermediate variable can simplify the code and reduce gas usage. Recommendation: bring variable currentOwner inside the loop, because its lifetime is limited to one iteration of the loop. Alternatively, below is a recommendation for more serious refactoring of the *executeTransaction* function:
1. introduce `mapping (bytes32 => int) public confirmationCounter`
2. increment `confirmationCounter` on “confirm”, decrement it on “revoke” (it does not exist, but could be useful to add - I will recommend it)
3. Instead of looping through the array of **_owners** and **indices**, simply check `confirmationCounter`
4. in the loop that checks signatures (**v**, **r**, **s** arrays), only count towards threshold if the recovered address has not confirmed that hash yet

Usability: I envisage a bit of a struggle if someone tries to use executeTransaction *function* without the associated frontend, but directly. Recommendation (only implementable after the refactoring recommended above): have a simplified version of executeTransaction without **v**, **r**, **s** parameters, it can be used as the existing Gnosis Wallet.

Existing Gnosis Multisig wallet supports revocation, but Gnosis Safe seems to lack this functionality. Recommendation: having personally used revocation myself, I recommend having it. It

Noted that *owners* array exists only for the convenience of the frontend. Actions are authorised based on *isOwner* mapping. Also, owners array is used for management of owners in the functions addOwner, removeOwner, replaceOwner. The code generated by Solidity compiler for array operations is more complex than the code for mapping operations. Therefore, use of arrays inflates generated byte code and makes byte code inspection less feasible. Recommendation: have a design discussion about pros/cons of having and not having owner array in the code. Possible alternatives:
- Add events to *setup*, *addOwner*, *removeOwner*, *replaceOwner* functions and have frontend to reconstitute the current list of owners from the events
- Change isOwner from a mapping to bool to mapping to address, effectively making it a linked list. In order to avoid edge cases and have a point to start, use sentinel technique:

```
mapping (address => address) public owners;
int ownerCount;
address public constant OWNER_SENTINEL = 0x1111; // Anything but 0

function setup() {
```

```
    owners[OWNER_SENTINEL] = OWNER_SENTINEL;
    ownerCount = 0;
}

function addOwner(address owner) {
    require(owner != 0);
    require(owners[owner] == 0);
    owners[OWNER_SENTINEL], owners[owner] = \
        owner, owners[OWNER_SENTINEL];
}

function removeOwner(address owner, address prev) {
    require(owners[owner] != 0);
    require(owners[prev] == owner);
    owners[prev], owners[owner] = \
        owners[owner], 0;
}

function replaceOwner(address oldOwner, address newOwner, address prev) {
    require(owners[oldOwner] != 0);
    require(owners[newOwner] == 0);
    require(owners[prev] = oldOwner);
    owners[prev], owners[oldOwner], owners[newOwner] = \
        newOwner, 0, owners[oldOwner];
}
```

Proxy.sol

In the fallback function, RETURNDATASIZE opcode is called 3 times. Recommendation: use intermediate variables inside the assembly in Proxy.sol contract, to avoid executing opcodes multiple times. Instead of this:

```
returndatacopy(0, 0, returndatasize())
switch success
case 0 { revert(0, returndatasize()) }
default { return(0, returndatasize()) }
```

have this:

```
let rds := returndatasize()
returndatacopy(0, 0, rds)
switch success
case 0 { revert(0, rds) }
default { return(0, rds) }
```

In the fallback invocation, the Solidity free memory pointer (address 64) can be overwritten. It is currently not used in the code, but it is good to know if more code is added in the future. Recommendation: place comment about it in the code.

If the delegatecall target (callee) throws instead of reverts, the output data will not be passed back to the proxy, and the returndatasize will return 0. Recommendation: There should be a test case covering this edge case.

DailyLimitExtension.sol

recommend to split into two extensions, one for Ether transfers, another - for token transfers.

SocialRecoveryExtension.sol

It allows a group of “friend” to replace any owner. It probably makes sense to only allow replacing a specific owner, otherwise these “friends” can take over the whole multisig.

General

the package.json does not list “solc” package, although it is required to run “truffle test”

Audit report for Gnosis Safe, iteration 2

This part of the report has been produced after reviewing a different codebase, which is a major refactoring of the previous one, and where most of the recommendations given in the interim report were implemented.

Although the changes in the repository were being made as the review was in progress (this was done to shorten the feedback loop and bring the completion closer), the final report refers to this commit of the code:
https://github.com/gnosis/safe-contracts/commit/942968d66a4fa200fe9757d02b377dbfc3c88636

This report only contains sections for the contracts with recommendations or observations.

Note on Solidity Optimiser and code generation

I have been informed that Gnosis Safe will be officially deployed as the code produced by solidity compiler without optimisations. Of course, since Gnosis Safe code is open source, anyone is free to compile it with optimiser on and deploy as such.
I do not hold strong opinion on the use or avoidance of Solidity optimiser. But I could not help noticing how inefficient is the code produced by Solidity compiler in the current version, at least without the optimiser. Therefore, I think it is worthwhile to try to create a hand-crafted byte code version of Gnosis Safe wallet with the same ABI. It might seem surprising, but it should be easier to audit such contract, as well as formally verify its properties.

Libraries

CreateAndAddModules.sol

The formulae for calculating the padded length of the data to 32 bytes was not correct:

```
i := add(i, add(0x20, mul(div(add(createBytesLength, 0x20), 0x20), 0x20)))
```

It has been corrected to:

```
i := add(i, add(0x20, mul(div(add(createBytesLength, 0x1f), 0x20), 0x20)))
```

I have verified in the byte code that this statement does indeed use CALL opcode to call “enableModule” instead of jump:

```
this.enableModule(module);
```

MultiSend.sol

This contract does not contain any functions to deposit Ether into it. Also, the only function it declares, “multiSend”, does not accept Ether. This leads me to believe that this contract (library) is only to be used in the context of “DelegateCall” operation in the multisig. No serious issues found. Suggestions:

1. Currently, the “transaction” parameter is a byte string that contains Call or DelegateCall operations packed according to ABI. This leaves some gaps filled with zeros. If these gaps were removed, the gas savings per operation could be around 400 gas. I understand that this encoding is used because of client side simplicity (there are ready available functions to perform ABI-style encoding of parameters).
2. Code inside the loop in the “multiSend” function could be improved by having “i” iterate over actual memory pointers and not offsets relative to “transactions”. A few ADD instructions would be spared and the code may become more readable.

Modules

I have been asked to only audit StateChannelModule.sol for the time being.

StateChannelModule.sol

The code is very similar to the code of GnosisSafePersonalEdition.sol, with 3 main differences:
1. No gas payback functionality
2. OwnerManager resides in another contract (addressable by “manager” state variable)
3. There is no “nonce” state variable.

No serious issues were found. Recommendations:

1. Use uint256 instead of uint8 for threshold (this will also require changes in the OwnerManager). There is no space or time advantage in using uint8 outside of tightly packed structs, in contrary, it adds extra code and gas costs during the execution. This change has been implemented and the effect was verified.
2. Use uint256[] instead of uint8[] as a type of “v” parameter to the “execTransaction” function. uint8[] does not give any space advantages, but leads to generating more code and costs more gas to execute. Value of uint256 will need to be explicitly converted to uint8 when calling ecrecover. This change has not been implemented, but instead an alternative solution has been created: all components of multiple signatures are now packed into a string of type “bytes”, taking 65 bytes per signature. Effects of this change has been verified.
3. Change the type “bool” in the “isExecuted” mapping to uint256. This will simplify generated byte code and save gas. This change has been implemented and the effect was verified.

Contracts

GnosisSafePersonalEdition.sol

No serious issues were found.
Recommendations:

1. Insert “revert” at the end of the “requiredTxGas” function, to prevent this auxiliary function to be used to actually effect transactions. This change has been implemented and effect was verified.
2. Adding “require” around token transfer in the “execAndPayTransaction” function. This is to support two semantically different implementations of ERC20 “transfer” function. Most common implementation is for the “transfer” function to throw or revert upon unsuccessful transfer. However, it is also possible to the “transfer” function to complete normally and return “false” as the result. This change has been implemented and effect was verified.
3. Use uint256[] instead of uint8[] as type for “v” parameter in the “execAndPayTransaction” function. This will simply the generated byte code and reduce gas usage, without requiring any extra space. Explicit cast of uint256 to uint8 for “ecrecover” function will be required. This change has not been implemented, but instead an alternative solution has been created: all components of multiple signatures are now packed into a string of type “bytes”, taking 65 bytes per signature. Effects of this change has been verified.
4. After recommendation (2) has been implemented, another potential issue has been found (which only exists due to historical reasons). In the first version of ERC20 “standard”, transfer

function did not have a return value. Expected semantics was to throw if the transfer is unsuccessful, and to return nothing if it was successful. Then, bool return value was added and semantic changed. But there are still token contracts out there (around 10% of token contracts) which still have old semantics. Solidity (new versions) now checks returndatasize when compiles “require”, and it will revert if the size is less than 32. Therefore, in the current implementation, it will not be possible to pay for execution with such old tokens. If this issue is to be fixed, the fix would require calling “transfer” function from within an assembly block. After the call, “returndatasize()” can be used to distinguish the cases of “transfer” returning nothing and it returning “false”. This change has been implemented and effect was verified.

Interesting observation - when compiling the statement “tx.origin.transfer(amount)”, the generated code gives extra 2300 gas to the CALL instruction only in the case when amount is zero. This is to fix the issue: https://etherscan.io/solcbuginfo?a=SendFailsForZeroEther
Another interesting observation - passing value more than 2 for “operation” parameter will cause transaction to throw (therefore consuming all available gas) and not revert, due to boundary checks placed by Solidity in all operations with enums.

GnosisSafeTeamEdition.sol

No serious issues were found.
Recommendations:

1. In function “checkAndClearConfirmations”, introduce an intermediate variable “approvals” equal to “isApproved[transactionHash]”, so avoid an extra Keccak256 computation inside the inner loop. This change has been implemented and effect was verified.
2. Use “uint256” instead of “bool” for the mapping “isApproved”. This eliminates an extra SLOAD instruction whenever the mapping is modified. This change has been implemented and effect was verified.
3. Remove the line that checked for double approval of the same hash (no security impact). This change has been implemented and effect was verified.
4. The same change as in the previous recommendation, but for “isExecuted” mapping. This change has been implemented and effect was verified.
5. Further gas cost reduction can be achieved by having the function “approveTransactionWithParameters” to accept ready transaction hash instead of computing it in the contract. This, however, adds more burden on the user interface to store the constituents of the hash somewhere else and broadcast them to all the owners of the safe.

Interesting observation - insertion of the “abi.encodedPacked” invocation before keccak256 (to quell solidity’s warning) adds an extra memory copy loop, which does not change semantics, but adds some extra code and gas costs.
Another interesting observation - passing value more than 2 for “operation” parameter will cause transaction to throw (therefore consuming all available gas) and not revert, due to boundary checks placed by Solidity in all operations with enums.

ModuleManager.sol

This contract contains two groups of functionality - management of modules and low-level functions that wrap CALL, DELEGATECALL and CREATE instructions.
Recommendations:

1. In function “disableModule”, add check for “module” actually being enabled. Otherwise it is possible to successfully execute this function with “prevModule” as some random address and “module” being 0. This does not pose safety/security risk now, but makes the function fragile in the face of future modifications. This change has been implemented and the effect was verified.
2. Consider remove “executeDelegateCall” function until there is a compelling use case for it. At the moment, the only case I see in the code is the use of libraries, like MultiSend. Combination

of this ability and modules effectively running in privileged mode opens up a wide variety of attacks via innocently looking modules.

Interesting observation - the byte code for the “getModules” function contains “CODECOPY” instruction which seems unusual. However, this is a cheap way of initialising memory with zeros.

OwnerManager.sol

This contracts maintains the circular linked list (suggested in my audit of the first iteration) of owners.
Recommendations:

1. Add checks to the “removeOwner” and “swapOwner” functions for the case when “prevOwner” is a random address, and “owner” and “oldOwner” are zero addresses. Although such calls require authorisation from the existing owners, they can damage the circular linked list invariants. This change has been implemented and the effect was verified.
2. Change type of “threshold” state variable from uint8 to uint256. There is no space advantage in using uint8 in this contract, and code generated for uint256 is shorter and more gas efficient. This change has been implemented and effect has been verified.