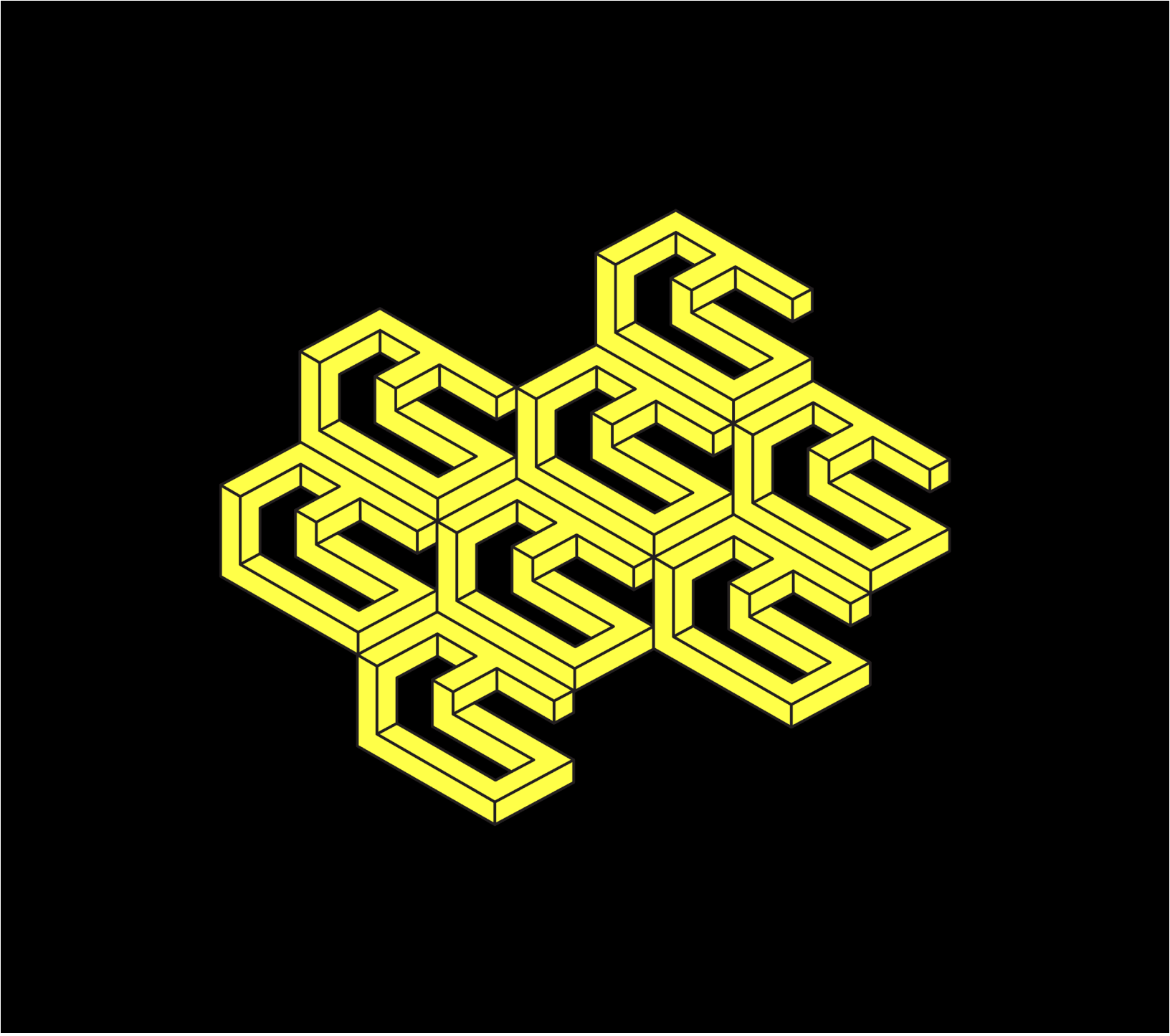

OpenGSN Smart Contracts Review

By ChainSafe Systems

January 2021 March 2021

ChainSafe

OpenGSN Smart Contracts Review

Auditor: Oleksii Matiiasevych

Warranty

This Code Review is provided on an “as is” basis, without warranty of any kind, express or implied. It is not intended to provide legal advice, and any information, assessments, summaries, or recommendations are provided only for convenience (each, and collectively a “recommendation”). Recommendations are not intended to be comprehensive or applicable in all situations. ChainSafe Systems does not guarantee that the Code Review will identify all instances of security vulnerabilities or other related issues.

Executive Summary

There were 3 critical, 5 major, 1 minor, 32 informational/optimizational issues identified in this version of the contracts. There are no known compiler bugs, for the specified compiler version (0.6.10), that might affect the contracts’ logic. Operation of the GSN protocol, especially in regards to being profitable for the relayers, depends on the effectiveness of the GSN nodes implementation, which was not reviewed during the course of this engagement. I enjoyed reading the code and all the parts were understood with ease. The exception was the gas economy part which in my opinion could use more commentary.

Update Verification Summary

There were 0 critical, 0 major, 0 minor, 9 informational/optimizational issues identified in the updated version of the contracts. There are no known compiler bugs, for the specified compiler version (0.7.6), that might affect the contracts’ logic. All the significant issues were addressed in this update, and 0 new issues were found.

1. Introduction

OpenGSN requested ChainSafe Systems to perform a review of the Gas Stations Network Protocol v2 contracts implementation by OpenGSN. The contracts in question can be identified by the following git commit hash:

`9856bdd7b55a779452fa7f30eebe39ef36a7b411 v2.0.1`

There are 21 contracts/libraries/interfaces in scope.

After the initial review, OpenGSN team applied a number of updates which can be identified by the following git commit hash:

`331166d9fcb5e01fd4dcf52e827123b7d3751972 release`

Additional verification was performed after that, results of which are included in the Appendix A.

2. Disclaimer

The review makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts for any specific purpose, or their bug free status. The review documentation below is for internal management discussion purposes only and should not be used or relied upon by external parties without the express written consent of ChainSafe Systems.

3. Executive Summary

There were 3 critical, 5 major, 1 minor, 32 informational/optimizational issues identified in this version of the contracts. There are no known compiler bugs, for the specified compiler version (0.6.10), that might affect the contracts' logic. Operation of the GSN protocol, especially in regards to being profitable for the relayers, depends on the effectiveness of the GSN nodes implementation, which was not reviewed during the course of this engagement. I enjoyed reading the code and all the parts were understood with ease. The exception was the gas economy part which in my opinion could use more commentary.

4. Critical Bugs and Vulnerabilities.

Three critical issues were identified during the course of review. Detailed explanations are provided in the Line By Line Review section. The short summary:

4.1. Using `_msgData()` inside of the GSN context abruptly ends contract execution.

4.2. Each RelayHub paymaster's entire balance could be drained immediately.

4.3 StakeManager stakers could be frontrun during their initial stake, which would result in their entire stake being drained.

5. Invariants Analysis With Fuzzing

Over the course of this review, a list of contract state invariants were constructed and tested with Echidna: A Fast Smart Contract Fuzzer

5.1. BasePaymaster. `relayHub` could only be changed by the owner.

5.2. BasePaymaster. `trustedForwarder` could only be changed by the owner.

5.3. BasePaymaster. Withdrawal could only be initiated by the owner.

5.4. BaseRelayRecipient. `msgSender` could only be modified if called by the `trustedForwarder`.

5.5. BaseRelayRecipient. `msgData` could only be modified if called by the `trustedForwarder`.

5.6. BaseRelayRecipient. `trustedForwarder` could not be modified.

5.7. Forwarder. `typeHashes` could not become `false` once being `true`.

5.8. Forwarder. `domains` could not become `false` once being `true`.

5.9. Forwarder. `nonces` could not decrease.

5.10. Forwarder. `nonces` could only be increased with a valid signature.

5.11. RelayHub. `minimumStake` could not be modified.

5.12. RelayHub. `minimumUnstakeDelay` could not be modified.

5.13. RelayHub. `maximumRecipientDeposit` could not be modified.

5.13. RelayHub. `maximumRecipientDeposit` could not be modified.

5.14. RelayHub. `gasOverhead` could not be modified.

5.15. RelayHub. `postOverhead` could not be modified.

5.16. RelayHub. `gasReserve` could not be modified.

5.17. RelayHub. `maxWorkerCount` could not be modified.

5.18. RelayHub. `stakeManager` could not be modified.

5.19. RelayHub. `penalizer` could not be modified.

5.20. RelayHub. `workerToManager` could not be cleared once assigned.

5.21. RelayHub. `workerCount` could not increase above the `maxWorkerCount`.

5.22. RelayHub. `penalize` could only be called by `penalizer`.

5.23. RelayHub. Paymaster's balance could not be decreased without the legitimate relay request. **Does not hold in current version.**

5.24. StakeManager. Relay manager and its owner cannot have the same address.

5.25. StakeManager. Sum of the `stakes` is always less than or equal to the StakeManager contract's balance.

5.26. StakeManager. Stake could not be decreased without a call from RelayHub or the stake owner.

5.27. StakeManager. `unstakeDelay` could not be decreased without a withdrawal.

6. Line By Line Review

6.1. Forwarder, line 40. Note, `suffixData` is always used as part of a hash preimage and never used in plain. Consider modifying the interface so that `suffixData` is accepted in a hash form.

6.2. Forwarder, line 63. **MAJOR**, the attacker could DoS executions by front running `execute` calls with not enough gas. Attack scenario:

- User signs a request with gas set to 1M.
- Relayer sends a request in transaction with a gas limit set to 1M.
- Attacker sends a transaction `Forwarder.execute(same request)` with a gas limit set to 50k and a gas price higher than Relayer.
- Attacker's transaction is mined first, call to the target contract goes Out Of Gas, while the user `nonce` in the forwarder gets updated.
- Relayer transaction fails too without being paid.

The most important thing here is that the user action will be cancelled. In this way, the attacker can practically stop the execution of any transaction through the forwarder as long as they have a budget for it. It is **not critical** though, as it can only delay the user action. A user can always send the transaction directly to the target, without using the forwarder at all.

One way to fix this problem is to validate available gas with `require(gasleft() >= req.gas);`

6.3. Forwarder, line 66. Note, there might not be enough gas left for value transfer, assuming that the attacker increased the forwarder balance after the relay transaction was already broadcasted. Relayers should supply enough extra gas for this situation.

6.4. Forwarder, line 73. Note, if the forwarder contract is deployed deterministically across different blockchains (having the same address) which use the same `chainId` then relay signatures could be replayed on them.

6.5. Forwarder, line 82. Note, partial validation doesn't guarantee valid request type will be produced. Consider removing it in order to optimize gas usage.

6.6. IForwarder, line 23. Note, outdated comment. Revert will also happen in case of an unknown domain separator or an unknown request type hash.

6.7. IForwarder, line 60. Note, outdated comment. A `typeSuffix` must not be empty. It should always have at least 1 additional parameter and must end with a ')'.

6.8. GsnTypes, line 8. Note, `gasPrice` must be validated inside of the paymaster's `preRelayedCall` in order not to overpay.

6.9. GsnTypes, line 9. Note, `pctRelayFee` must be validated inside of the paymaster's `preRelayedCall` in order not to overpay.

6.10. GsnTypes, line 10. Note, `baseRelayFee` must be validated inside of the paymaster's `preRelayedCall` in order not to overpay.

6.11. IKnowForwarderAddress, line 4. Note, this interface is only used for testing purposes. Consider removing it.

6.12. IRelayHub, line 10. Note, throughout the file, style mismatch is noticed as double slash vs triple slash comments.

6.13. IRelayHub, line 134. Note, `paymasterMaxAcceptanceBudget` is missing in the comments section documenting this function.

6.14. IRelayRecipient, line 32. Note, the comments section ends abruptly with an unfinished explanation.

6.15. IStakeManager, line 55. Optimization, the `unstakeDelay` and `withdrawBlock` can be packed into the same slot as the `owner` in order to optimize gas usage.

6.16. GsnEip712Library, line 146. Note, the order of fields is different from the `RelayData`. The `forwarder` is located at the end of the struct, while it is not at the end in the preimage.

6.17. GsnEip712Library, line 151. Note, excessive blank lines.

6.18. MinLibBytes, line 22. Note, the function `readAddress` is not used. Consider removing it as obsolete.

6.19. BasePaymaster, line 98. Note, Ethereum users are famous for sending ERC20 tokens to any address they see. Consider adding a function to withdraw such tokens from the paymaster, in case they end up locked.

6.20. BaseRelayRecipient, line 29. **MAJOR**, `_msgSender` will fail to recognize a relayed call in case of original data being empty or shorter than 4 bytes (such that could be used in a fallback function). This will restrict some contracts from integrating GSN seamlessly. This limitation could be lifted by changing the condition to `msg.data.length >= 20`.

6.21. BaseRelayRecipient, line 50. **MAJOR**, `_msgData` will fail to recognize a relayed call in case of original data being empty or shorter than 4 bytes (such that could be used in a fallback function). This will restrict some contracts from integrating GSN seamlessly. This limitation could be lifted by changing the condition to `msg.data.length >= 20`.

6.22. BaseRelayRecipient, line 61. **CRITICAL**, the contract execution ends here.

- Any function that will make use of `_msgData` in GSN context will be abruptly ended, because assembly's `return` expression works differently from Solidity's return statement.
- Bytes encoding in memory is incorrect for the internal context. Currently the final layout is `<offset> <length> <bytes>` while the correct layout is just `<length> <bytes>`.
- The free memory pointer located at `0x40` must be updated at the end of the assembly block, otherwise memory corruption might occur in the code that called `_msgData`.

6.23. BatchForwarder, line 28. Note, outdated comment.

6.24. Penalizer, line 56. Optimization, `abi.encodePacked` is excessive here, because it's only argument already has a `bytes` type.

6.25. Penalizer, line 57. Optimization, `abi.encodePacked` is excessive here, because it's only argument already has a `bytes` type.

6.26. Penalizer, line 99. Optimization, `abi.encodePacked` is excessive here, because it's only argument already has a `bytes` type.

6.27. RelayHub, line 26. Optimization, many variables from this batch could be packed together to take less storage space and make future reads cheaper.

6.28. RelayHub, line 79. Optimization, `workerCount[relayManager]` read thrice in this function. Consider putting it into a local variable to optimize gas usage.

6.29. RelayHub, line 111. Optimization, `balances[account]` read twice in this function. Consider putting it into a local variable to optimize gas usage.

6.30 RelayHub, line 179. **MAJOR**, `MinLibBytes.readBytes4` will fail in case of request data being empty or shorter than 4 bytes (such that could be used in a fallback function). This will restrict some contracts from integrating GSN seamlessly.

6.31. RelayHub, line 240. Note, subtraction could cause an underflow here, though it is safe because it will revert further down in `calculateCharge` and the worker can be subsequently penalized.

6.32 RelayHub, line 244. Optimization, `workerToManager[msg.sender]` is read six times in this function. Consider putting it into a local variable or memory to optimize gas usage.

6.33. RelayHub, line 220. **CRITICAL**, `vars.innerGasUsed` can be manipulated by the relayer in various ways to bypass `acceptanceBudget` condition and drain any paymaster's balance in a single transaction. Manipulation is achieved by setting one of the following `relayCall` parameters to waste data of various length:

- signature
- approvalData
- relayRequest.relayData.paymasterData
- relayRequest.request.data

Relayer will then set the relayData.baseRelayFee to the current balance of the paymaster, relayData.gasPrice to zero, and relayData.paymaster to the paymaster's address. Other parameters don't have much relevance in this case. The relayed call will fail, but acceptanceBudget will be reached, so paymaster will end up paying the whole balance to the relayer.

6.34. RelayHub, line 323. **MAJOR**, reverting anywhere inside of the relayed call results in a forwarder signature of the sender being left unused and publicly available for prolonged periods of time. This signature could be used by an attacker if the user stops using the protocol and does not make sure to execute or invalidate it. Consider the following attack scenario:

- User signs a call to transfer 1000 GSN enabled ERC20 tokens to address X, while having only 50 on balance.
- Call fails (and the nonce left unused in the forwarder).
- User stops using the GSN and proceeds with using his wallet directly.
- Years pass and eventually the user has 1000+ of the same tokens on his balance.
- Attacker sends a transaction to the forwarder with request from step #1, forcing the user to do the transfer of 1000 tokens.

This issue could be mitigated by introducing an expiration date for the requests.

6.35. RelayHub, line 389. Note, the `stakeInfo` is excessively pulled from the `StakeManager` and then immediately passed back to it. Consider fetching `stakeInfo` inside of the `penalizeRelayManager` call.

6.36. StakeManager, line 32. Minor, an attacker could disable the `stakeForAddress` function for any `relayManager` by making its `owner` a relay manager for themselves. They will need to call `stakeForAddress(targetOwner, 0)` and `targetOwner` will be restricted from increasing their stake.

6.37 StakeManager, line 33. **CRITICAL**, a frontrunning attack could be executed to burn half of the stake of the new relay manager and to steal the second half. The attack scenario:

- User A calls `stakeForAddress{value: 1 ETH}(newManager, newDelay)`
- User B front runs A with a call to a malicious contract that does the following:
  - `stakeForAddress(newManager, 0);`
  - `authorizeHubByOwner(newManager, Bob);`
  - `unlockStake(newManager);`
  - `withdrawStake(newManager);`
- Now A's transaction mines successfully, and the `newManager` authorized B to be a `relayHub`
- B calls `penalizeRelayManager(newManager, Bob, 1 ETH)`
- B gets 0.5 ETH, A loses the whole stake.

6.38. StakeManager, line 35. Note, the `unstakeDelay` could be accidentally set to unrealistically long periods of time resulting in a forever locked stake. Consider limiting this parameter.

6.39. StakeManager, line 36. Optimization, there are 3 excessive storage reads here, consider using local variables instead to optimize gas usage.

6.40. StakeManager, line 44. Optimization, `info.withdrawBlock` could be stored in the local variable to optimize excessive reads.

6.41. StakeManager, line 133. Note, burning ether could be wasteful. Consider directing it to some cause instead.

Oleksii Matiiasevych

Appendix A.

OpenGSN Smart Contracts Update Verification

Auditor: Oleksii Matiiasevych

1. Executive Summary

There were 0 critical, 0 major, 0 minor, 9 informational/optimizational issues identified in the updated version of the contracts. There are no known compiler bugs, for the specified compiler version (0.7.6), that might affect the contracts' logic. All the significant issues were addressed in this update, and 0 new issues were found.

2. Remaining Insignificant Issues

2.1. Forwarder, line 40. Note, `suffixData` is always used as part of a hash preimage and never used in plain. Consider modifying the interface so that `suffixData` is accepted in a hash form.

2.2. IStakeManager, line 55. Optimization, the `unstakeDelay` and `withdrawBlock` can be packed into the same slot as the `owner` in order to optimize gas usage.

2.3. MinLibBytes, line 22. Note, the function `readAddress` is not used. Consider removing it as obsolete.

2.4. BasePaymaster, line 99. Note, Ethereum users are famous for sending ERC20 tokens to any address they see. Consider adding a function to withdraw such tokens from the paymaster, in case they end up locked.

2.5. RelayHub, line 121. Optimization, `balances[account]` read twice in this function. Consider putting it into a local variable to optimize gas usage.

2.6. RelayHub, line 270. Note, subtraction could cause an underflow here, though it is safe because it will revert further down in `calculateCharge` and the worker can be subsequently penalized.

2.7. RelayHub, line 418. Note, the `stakeInfo` is excessively pulled from the `StakeManager` and then immediately passed back to it. Consider fetching `stakeInfo` inside of the `penalizeRelayManager` call.

2.8. StakeManager, line 43. Optimization, there are 3 excessive storage reads here, consider using local variables instead to optimize gas usage.

2.9. StakeManager, line 140. Note, burning ether could be wasteful. Consider directing it to some cause instead.