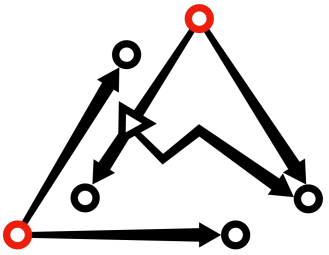

Least Authority

PRIVACY MATTERS

Gas Station Network (GSN), Paymaster Contracts, + Forwarder Contract

Security Audit Report

Ethereum Foundation

Updated Final Report Version: 27 December 2020

Table of Contents

Overview

Background

The Ethereum Gas Station Network has requested that Least Authority perform a security audit of the Gas States Network (GSN), the Paymaster Contracts, and the GSN Forwarder Contract, supported by the Ethereum Foundation

- GSN: GSN is a decentralized system that aims to improve decentralized applications usability without sacrificing security. GSN abstracts away gas to minimize onboarding and UX friction for decentralized applications. With GSN, gasless clients can interact with Ethereum contracts without users needing ETH for transaction fees.
- Paymaster: In order to cover their expenses, the transaction costs will be charged from a balance of a special contract, called Paymaster.
- Forwarder: The GSN Forwarding Contract is the generic meta transaction forwarding contract proposed as an ERC 2770. The contract verifies a signature and appends msg.sender to the last 20 bytes of msg.data, allowing recipient contracts that trust the forwarder to accept native meta transactions. The intended use of the forwarder contract is to be a neutral singleton on-chain that any contract can leverage to support meta transactions.

Project Dates

- **July 30 - August 19**: Code review *(Completed)*
- **August 21**: Delivery of Initial Audit Report *(Completed)*
- **December 7 - 9:** Verification *(Completed)*
- **December 11:** Delivery of Final Audit Report *(Completed)*
- **December 27**: Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Nathan Ginnever, Security Researcher and Engineer
- Dylan Lott, Security Researcher and Engineer
- Dominc Tarr, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the GSN, the Paymaster Contracts, and the GSN Forwarder Contract followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:

- GSN: https://github.com/opengsn/gsn
- Paymaster Contracts: https://github.com/opengsn/gsn-paymasters
- GSN Forwarder Implementation: https://github.com/opengsn/forwarder

Specifically, we examined the Git revisions for our initial review:

GSN: `3bc59af87bed764cedb828dedbb3a18188b21b0a`

GSN-Paymasters: `a7a28e93bac235fde3c177df3b4e5753a7004afb`

Forwarder Contracts: `bf3447b2e54c407a6e02e0bd2af858fe71e13711`

For the verification, we examined the Git revision:

GSN: `d0deb33e851d069c4c33cbff5183fc1a7ff8440c`

GSN-Paymasters: `dd6d6c038cecc067f9141c5927b4ced2d302a4f4`

Forwarder Contracts: `d0deb33e851d069c4c33cbff5183fc1a7ff8440c`

All file references in this document use Unix-style paths relative to the project's root directory.

Supporting Documentation

The following documentation was available to the review team:

- README: https://github.com/opengsn/gsn/blob/master/README.md
- GSN Documentation: https://docs.opengsn.org/learn/index.html
- Paymaster Documentation: https://docs.opengsn.org/contracts/#paymaster
- Kovan Testnet: https://dashboard.tenderly.co/contract/kovan/0x77777e800704fb61b0c10aa7b93985f835ec23fa
- ERC Drafts
  - EIP 1613 (GSN v2 is isomorphic in both form of function to the GSN v1): https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1613.md
  - EIP 2770 Forwarder Singleton: https://github.com/forshtat/EIPs/blob/forwarder_eip/EIPS/eip-2770.md
  - EIP 2771 (WIP) - Secure Native Meta Transaction Recipient: https://docs.google.com/document/d/1UXNctr5TPKRAGP1wpMQhX69AgCM5jve2TFSrDfFaCo8/edit
- Internal GSNv2 Documents
  - Security Model: https://www.notion.so/GSN-v2-security-model-4115f7e4d067413d9d18a0a3f6086c1a
  - GSN Replay DDoS Attack and Mitigation: https://www.notion.so/GSN-Replay-DDoS-attack-and-mitigation-a99ffea3cb8e49c986d76124882072d6
  - Trust Model: https://docs.google.com/document/d/109abcg0szQRKNzzwIZQmHHyowlCYhvr30b3F3Fclt5o/edit
  - Relay Flow: https://docs.google.com/document/d/1Du6l8Yx7GbIK-vec7ypN6kb-cCHcFzJLNYYtXhk9w3w/edit#heading=h.9kfxcqxwsjc0
  - Beta Paymaster Model: https://docs.google.com/document/d/1Feec_OqAXEqanDRsKQ-NtBz1kpcWskFhD7X-b5GhcM4/edit
  - Paymaster API: https://docs.google.com/document/d/1Up6PqrEkdMB9n7VAJO03ozZS13L-OtaqZjtZVd7tgY0/edit
  - Relay Server Auto-Upgradability Model: https://docs.google.com/document/d/1AANPUBP5JgSwYDywYvTSrCSWM5oguKCj1xUTZk8Afkl/edit#heading=h.z5aqgmfbykl9

Areas of Concern

Our investigation focused on the following areas:

- Protocol:
    - Integrity of forwarder recipients: sender and nonce cannot be spoofed for any recipient that trusts the forwarder;
    - Griefing mitigations mechanisms are sound in that the asymmetries are tilted to the benefit of the defense;
    - Availability guarantees provided by the mechanism design of the protocol have not regressed significantly since GSN v1 / EIP 1613 (GSN v2 is isomorphic in both form of function to the GSN v1);
- All Contracts:
    - Integrity of deposits by GSN participants: all third party deposits by relay servers and Paymasters can not be stolen or frozen;
    - Assessing the security of the Paymaster templates;
    - Correctness of the implementation;
    - Adversarial actions and other attacks on the network;
    - Potential misuse and gaming of the smart contracts;
    - Attacks that impact funds, such as the draining or the manipulation of funds;
    - Mismanagement of funds via transactions;
    - Economic incentives: ensure token economics (monetary incentives to punish bad behavior and reward good behavior) are included and functional;
    - DoS/security exploits that would impact the contracts intended use or disrupt the execution of the contract;
    - Vulnerabilities in the smart contracts code;
    - Protection against malicious attacks and other ways to exploit contracts;
    - Inappropriate permissions and excess authority;
    - Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

The GSN team was an invaluable resource throughout the audit and was readily available to answer questions and provide further context on specific areas of concern. It is clear from the project documentation, code, and our interactions with the GSN team that security has been strongly considered and of the utmost priority throughout the construction. We commend the GSN team for their diligence and efforts.

Despite this clear effort towards security, there are a few suggestions and issues that were identified during the audit, as noted below.

Review Scope

During our review of the Gas Station Network, our team closely evaluated the core GSN contracts, including the RelayHub, StakeManager, Penalizer and the Paymaster base class, in addition to the Paymaster and Forwarder contracts. The scope was comprehensive, covering most components of the system, and included a review of the GSN protocol. However, the design of the relay server auto-upgradeability mechanism was considered out of scope for the review.

Code Quality + Documentation

Our team found the code to be clean, well organized, and of high quality, in that it adhered closely to development best practices. The GSN system relies heavily on interfaces and have created a set of interfaces for creating a meta transaction's setup for numerous use cases. Thus, they have created a framework for meta transactions that can be extended and inherited from, in order to include a large variety of trust models in a manner that can be reasonably secure for all participants.

In addition, we found the Solidity code to be considerably advanced in most areas of the codebase, demonstrating the GSN team's deep understanding of Solidity and the Ethereum Virtual Machine (EVM). The GSN team employs advanced usage of deep language features such as the recently experimental ABI encoder V2 to support array data, which has only been ready for deployment in production since Solidity compiler v0.6.0. Usage of modifiers, inheritance, and gas saving techniques, such as comparing hashed data rather than the data itself, demonstrates a comprehensive understanding of Solidity and we commended the GSN team for employing relatively new and experimental techniques carefully such that the security continues to be a primary concern.

The codebase includes high-level comments in several areas, particularly describing the design rationale for more complicated functions. We found the interface contracts to be particularly well commented and follow style guides, which was useful in determining how the various components may potentially interact.

There are other areas in the code, however, that would benefit from additional comments. While there are comments providing a description of functionality, they do not adhere to the Solidity standard style guidelines which would include documenting parameters. Detailed context on how a single component works within the rest of the system would provide further guidance to users and reviewers of the code. For example, the utilities library implementing EIP 712 contains only one commented function. As a result, we suggest that comments be integrated into the codebase at large, particularly in areas of increased complexity (Suggestion 1).

The codebase also includes a considerable test suite that includes happy path tests, including regression tests and tests for specific malicious actions, in an attempt to prevent malicious activity, which exemplifies a strong focus on security and safety by the GSN team.

The existing project documentation is comprehensive with a strong emphasis on security. The documentation proves to be useful in that it covers certain potential vulnerabilities, with a heavy focus on different attack vectors and mitigations. However, the documentation is not yet published and we suggest compiling the informal documentation provided to our team on attacks into the formal GSN documentation. In addition, documentation around the expected use cases would prove to be helpful. Since the GSN codebase largely comprises a framework for handling meta transactions, we recommend having more documentation available describing the expected use cases and how the system handles requests from beginning to end (Suggestion 1).

System & Mechanism Design

GSN alters the Ethereum blockchain protocol to allow meta transactions, which shifts the trust model of who pays for the computational resources of a blockchain network. Meta transactions are a desirable feature for many reasons, as they increase usability and lower barriers to entry into blockchain networks. However, allowing for a separation of payment concerns introduces a third party to the transaction. This creates a fundamental trust problem that the GSN does a notable job of mitigating. Users must trust that the relayers will not deny service by broadcasting the same nonce and a relayer trusts that a Paymaster will reimburse the transaction, while the Paymaster needs to trust that users and relayers will not expend too many resources and deplete the funds in the Paymaster contracts.

All of these concerns have been addressed by some mechanism in the GSN network, including stake for relayers to be penalized when incorrectly formatting transactions, forwarders ability to be trusted by the application such that any application is not exposed to the risk of the open GSN relay network, and the limits placed on Paymasters for what they will pay out to prevent malicious usage of gas funds. The GSN team brought to our attention notes on attacks relating to these trust problems, which we examined carefully and have raised edge case concerns in Issue D and Issue E.

Our team found that most of the issues in the system design exist in the trust relationship between Paymasters and relay servers (Issue C). The incentives between this part of the system must be examined and weighed carefully and any changes to these components should be carefully considered, internally tested, and followed up by a third party review (Issue E).

The penalty system also requires a careful review of how its incentives line up with Paymasters and relays, since it is meant as a check against poorly behaving relays. Penalties must be administered carefully to keep incentives aligned between the other components in the system.

Specific Issues

We list the issues we found in the code in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Relay URL Filter Does Not Remove Duplicates	Resolved
Issue B: RelayClient Does Not Timeout Relay Request	Resolved
Issue C: Malicious Paymaster	Resolved
Issue D: Relay Server Griefing (Known Issue)	Resolved
Issue E: Relay Penalizer Incentive Misalignment	Resolved
Issue F: Attacker May Request Block Gas Limit Worth of Gas	Resolved
Issue G: RelayHub Makes Unguarded Call to Untrusted Contract	Resolved
Suggestion 1: Improve Documentation	Partially Resolved
Suggestion 2: Consider Implementing an Approve / Confirm Owner Transfer	Resolved
Suggestion 3: Complete Paymaster Gas Calculation	Partially Resolved

Issue A: Relay URL Filter Does Not Remove Duplicates

Location

https://github.com/LeastAuthority/gsn/blob/3bc59af87bed764cedb828dedbb3a18188b21b0a/src/relayclient/RelaySelectionManager.ts#L97

Synopsis

When a relay client attempts to select the next relay, it filters through a list of given relays. This filter should remove duplicates, as stated in the comments for the `RelaySelectionManager`. However, the code performing the filtering does not remove duplicates.

Impact

Minor. The comments state that the filtering should occur and that the relays URLs are used as keys in maps. As a result, these duplicates may cause data to be overwritten elsewhere in the client.

Preconditions

Duplicates would have to be returned from the transaction details that are passed into the `RelaySelectionManager`.

Technical Details

The array `` `slice` `` is acted on by the `` `filter` `` method, however, the return value of the operation is never assigned to anything and never returned as a result.

Remediation

The value returned from the `` `.filter()` `` method performed on `` `slice` `` should be assigned to a new value and that value should be returned. An example of the problem and the fix is available. Additionally, we recommend adding a regression test for this case.

Status

The GSN team has responded that while the code does not rely on the filter, the filter has been removed.

Verification

Resolved.

Issue B: RelayClient Does Not Timeout Relay Request

Location

https://github.com/LeastAuthority/gsn/blob/master/src/relayclient/HttpWrapper.ts#L10-L12

Synopsis

After a client finds a relay server that responds to a ping successfully, it does not expect the next request to timeout.

Impact

A malicious relay server can stall any client that makes a request to it by responding to the ping quickly but stalling on the relay request.

Preconditions

The client has not specifically configured a timeout.

Technical Details

GSN uses the Axios library to perform http requests, which defaults to not using timeouts on requests. The `RelayClient` does not configure a timeout and requests are made without a timeout as a result.

The Axios library is initialized with a configuration object, which does not have a timeout property by default. The timeout property can also be passed as the third argument to any call, which it currently is

not. The Axios library is also wrapped by `HttpWrapper`, which is then wrapped by `HttpClient`. `HttpClient` is created by `GSNConfigurator` in `getDependencies`, which is called by `RelayClient`.

Before selecting a `RelayServer` to use, the `RelayClient` pings several `RelayServers`. The ping request is also made without a timeout property. However, since multiple pings are made in parallel, it will work as long as at least one `RelayServer` does not stall.

Remediation

Make timeout part of the default http configuration.

Status

A 15-second default timeout setting has been implemented for the `HttpClient`.

Verification

Resolved.

Issue C: Malicious Paymaster

Location

https://github.com/LeastAuthority/gsn/blob/3bc59af87bed764cedb828dedbb3a18188b21b0a/src/relayserver/RelayServer.ts

Synopsis

This issue is related to the gas depletion attacks described by the GSN team.

A carefully designed Paymaster appears to accept a relay when run as a view function, however, it rejects the transaction on-chain. This causes the `RelayServer` to pay for all gas up to the acceptance budget.

Impact

An attacker could drain the relay servers of their budget by making many requests to the malicious Paymaster. There is no way to disable supporting a particular Paymaster in the current `RelayServer` code, as a result, all `RelayServers` are vulnerable.

Preconditions

The attacker would need a malicious Paymaster contract and the ability to request to relay servers.

Feasibility

This attack would be straightforward with a suitable Paymaster contract.

Technical Details

The attacker first deploys a contract that will accept a relay request when it is run off-chain, but rejects it when it is run on-chain. When making a request, the relay server will first run the Paymaster's accept function off-chain and if it rejects the transaction, it will not submit it to the blockchain. If the Paymaster accepts it off-chain, but rejects it on-chain, the relay server is left to pay the gas for that transaction.

A contract that runs one way off-chain and another way on-chain can be created by exploiting the default ordering of transactions within a block. Both Parity and Geth sort transactions within a block by gas price, with the highest first and without reordering transactions from a single address.

A contract is made that has two modes, accept and reject. The mode can only be switched by the malicious contract owner. First, the mode is set as accept, then many requests with lower gas prices are

made to many `RelayServers`. The malicious Paymaster owner intends for these transactions to take several minutes to be confirmed. The malicious Paymaster then quickly sends a transaction to switch the Paymaster mode to reject. The mode switch transaction is sent with a much higher gas price so that even if it is mined in the same block as one of the relay requests, it will be sorted to the start of the block and thus mined before the relayed transaction. As a result, the relayed transaction is run in reject mode. The attacker waits until all pending transactions are confirmed and then switches back to accept mode for the next round of attack.

To have maximum impact, when run in accept mode the contract would use very little gas, but in reject mode it should loop until all available gas has been used up.

Mitigation

At present, there is no way for a relay server operator to disable support for a particular Paymaster. However, because relay servers are just servers and not contracts, a change could be deployed rapidly. We recommend that the ability to disable particular Paymasters be added. An even safer alternative would be to explicitly enable Paymasters and then audit the Paymasters first.

Remediation

A possible approach would be to limit the on-chain function to run with exactly the gas used by the off-chain run. However, this creates an easy way for the malicious Paymaster to detect if it is running on-chain or not, reducing the impact of the attack, but also making it easier to execute. It is not possible to have Ethereum lie to the contract about the gas remaining, but it would be possible to statically analyze the contract and detect that it does not access gas remaining. If not, then it cannot be using this attack. This option would also have the downside of likely reverting when run on contracts that did not have exactly the same gas used, a condition that might be triggered by the state of the contract changing due to other calls to the same contract being run and mined in between the time of the off-chain run and the on-chain run.

It has also been suggested during discussions with the GSN team to have a reputation mechanism where relays would automatically block Paymasters that reject on-chain. This will likely introduce new edge cases and could possibly be attacked by calling the Paymaster without running the check off-chain first.

We do not recommend interacting with arbitrary contracts without the ability to disable specific instances and the GSN community should be encouraged to audit Paymaster contracts.

Status

The relay client will still relay transactions to arbitrary Paymasters, but will track the number of reverts, and will throttle requests to a Paymaster if it reverts a certain number of times within a window. If it continues to error, it will eventually be permanently blocked. While this approach does not prevent malicious paymasters, it does limit their impact on a particular relay server. It is worth noting, however, that each relay server keeps track of the failed requests locally, so the impact of the attack is on every relay server. As a result, an attack would have greater impact if there are many relay servers in the network. Once a malicious Paymaster is blocked by a relay, the attacker would need to deploy another Paymaster contract, thus making the attack more expensive. It seems likely that this implemented mitigation is sufficient enough to discourage the attack in practice due to the increased cost.

We recommend taking an additional step to provide the ability to manually allow or disallow particular Paymasters. If a currently unknown attack is discovered, its impact will be greater without a pre-existing ability to disable the specific interaction that enables it.

Verification

Resolved.

Issue D: Relay Server Griefing (Known Issue)

Location

https://github.com/LeastAuthority/gsn/blob/master/contracts/Penalizer.sol

Synopsis

The client submits the same relayed transaction to many relay servers, which runs successfully off-chain, but will likely only run successfully on the first relayed transaction in order to get mined. Other relay servers who submit it will be left paying the gas of the failed transaction.

Impact

This will result in increased cost of running a relay server and it could potentially be an attack on the relay network.

Preconditions

A contract supporting GSN that is likely to run a transaction once, but not twice.

Feasibility

This attack is straightforward and also has the potential to take place accidentally.

Technical Details

The relay server receives a signed transaction from a client and then checks if it is valid by a number of checks, including running the transaction off-chain. If it is accepted, it then submits it to the blockchain. However, if the same transaction has been submitted to multiple relay servers, they will each pass the off-chain check, but the first on-chain transaction will likely change the state of the contract so that the second and subsequent runs revert. This will leave the other relay servers to pay for gas on the failed transaction.

Mitigation

The GSN team explained that they intend to handle this case by relay servers detecting reverting transactions and introducing a random delay, similar to the random delays used to prevent collisions in network protocols. During the delay, the relay server would have time to check for pending transactions of the same call, and if a duplicate is detected, it would reject the transaction. This should greatly reduce the cost to relays without stopping it entirely.

Alternatively, there are cases where a client may need to legitimately re-request that a transaction be run by another client. For example, this may occur if their transaction stalls because a relay server goes offline after submitting a transaction with insufficient gas and does not increase the gas price again. To avoid any relay refusing to rerun their transaction, the client could alter subtle aspects of the transaction so that it does not hash the same (such as fractional token amounts). However, this would also have the effect of breaking the mitigation.

Another potential approach, which could be used as well as the above, would be to temporarily increase the relay fee to cover costs.

Remediation

A possible near-full remediation would be to develop a way to run a view function that takes into account possible state change for all pending transactions. It would take into account both the current confirmed state of the chain and different possible selections of pending transactions. To our knowledge, this does not yet exist. While it would be a valuable service to the Ethereum community, it would require a significant development effort. In addition, it still would not guarantee that when a relay sends a

transaction that it will not collide with another one, including one that may not yet have been sent. This is an unfortunate artifact due to the design of Ethereum.

Status

The suggested mitigation has been implemented and a delay will be triggered if the attack is attempted once. Additionally, the GSN team has implemented the ability for a relay server to temporarily increase fees to cover costs.

The GSN team has also responded that the cost of an attack is non-zero, thus reducing the cost-effectiveness of an attack and thus discouraging the attack due to the increased cost.

Although the GSN team has taken the recommended steps to mitigate this issue and it is resolved per this report, there are remaining concerns given the complexity around this issue. As a result, we believe it should be closely monitored to ensure the implemented remediation tactics are effective. Specifically, we suggest a further attack analysis on the cost of the fee as compared to the opportunity cost griefed on the relayers, and how these costs translate over time with fluctuating value of the fee / gas prices.

Verification

Resolved.

Issue E: Relay Penalizer Incentive Misalignment

Location

https://github.com/LeastAuthority/gsn/blob/master/contracts/Penalizer.so

Synopsis

This issue raises the question of who is watching the relayers. The penalizer provides a mechanism to discourage a Denial of Service (DoS) attack, however, the incentives and protocol for submitting fraud proofs to the penalizer contract are not clear.

Impact

If undetected or unreported, relayers may abuse users by denying their transactions from updating the application state.

Preconditions

Those watching for invalid transactions must be rationally deciding that they will not receive a reward and must stop watching for invalid relayer transactions that may be attacking a user. This may require client software and protocol changes that would undo functionality provided by default to aid in the search for invalid transactions.

Feasibility

Given that there is a challenge in the penalizer contract with stake on the line to discourage this behavior, this is unlikely to happen and would require the relayers infraction going unnoticed. As a result, one must assume that going unnoticed is a possibility.

Technical Details

The relayers may DoS users by broadcasting a double nonce such that any further transactions are not able to make their way into the application. A further description of the attack can be found in the GSN Replay DDoS Attack and Mitigation document provided to us by the GSN team. In general, a relayer may broadcast an incorrectly formatted transaction in the fields of gas limit and bad method signature. It is less clear when this infraction would be beneficial for an attacker and we suggest creating

documentation (Suggestion 1) providing the reasoning for mitigating these transaction faults with severe penalty. The penalty for a relayer committing either of these infractions is removal from the hub and total loss of stake. Given that this is a severe punishment, it is reasonable to assume that the events will happen infrequently or with extremely low probability. Furthermore, given the low probability of success, we theorize that rational actors in the GSN will choose not to scan for invalid transactions as a result.

There is an initial incentive for any user that is denied service to punish a relayer. However, it is not clear who will know about the infraction or if the user will be able to reach the network to punish the relayer.

Mitigation

This calls to question systems like watch towers or pieces of software with the sole purpose of scanning for infractions. This is a complicated area of research and it is yet unclear to our team which watch tower system is best for all systems. The GSN team has plans for some type of watch tower in the form of a penalizer process possibly running on cloud infrastructure for high up time, and plans to initially implement full node relayers with scanning code to altruistically scan the transaction mempool. We encourage the GSN team to monitor developments in this area of research.

Status

The GSN team has implemented a new feature that has relay clients doubling as transaction verifiers per the suggested mitigation. The GSN team has also responded that the relayers are incentivised to report infractions given that there is stake to claim if one is found.

Although these mitigations sufficiently address the stated issue, we encourage the GSN team to continue researching this incentive. If an infraction is not likely to happen frequently, and most nodes are found to have turned off the code that altruistically searches for infractions, there could be no real incentive for any node to report issues under a rational assumption that it costs more to look for infractions without a high likelihood of reward.

Verification

Resolved.

Issue F: Attacker May Request Block Gas Limit Worth of Gas

Location

https://github.com/opengsn/gsn/blob/master/contracts/BasePaymaster.sol#L36

Synopsis

This issue amplifies the attack we describe in Issue C with malicious Paymaster contracts. Neither the relay server nor `RelayHub` enforce a maximum gas limit against a Paymaster, but will relay transactions as long as the Paymaster has sufficient balance to cover. As a result, the Paymaster controls the exposure of the relays. In general, the initiators of relay requests control the maximum amount of Ether a relayer will burn in a single transaction up to the block gas limit.

Impact

Combined with a malicious Paymaster (Issue C), the balance of a relay (or all relays) can be rapidly burned.

Preconditions

The attacker would need a malicious Paymaster contract and enough balance to appear able to cover the max cost.

Feasibility

This attack would be straightforward (see Issue C).

Technical Details

The `BasePaymaster` defines a maximum acceptance budget as a static public variable `PAYMASTER_ACCEPTANCE_BUDGET`, but this is only checked at compile time. As a result, a different Paymaster contract can be used as long as it follows the same interface. The `RelayHub` checks that the Paymaster has enough balance to cover the transaction, however, it does not check that it is within a reasonable limit. For each request made to a relay server, it will calculate a constant `maxPossibleGas` used to determine how much gas the relayer needs to provide in order to ensure that the transaction is successful. The client sends a `gasLimit` to the relay server to cover the execution costs of the forwarder transaction and the relay server will contact the Paymaster contract to add enough gas based on the value of the acceptance budget. The relay server may be attacked and use the maximum available gas in an Ethereum block during the attack.

Remediation

The `RelayServer` needs a maximum gas limit that it is willing to relay or specifically reject relay requests to Paymasters that return a `PAYMASTER_ACCEPTANCE_BUDGET` that is too high for the exposure tolerance of the relay.

Status

During the time in which our team audited an earlier version of the code, the GSN team merged a pull request that limits the `RelayServer's` exposure to a malicious Paymaster. As a result, a maximum acceptance budget is passed to the `RelayHub`. The `RelayHub` can still be griefed by a malicious Paymaster but the exposure is limited. However, the fix was vulnerable to Issue G, but that has now been also resolved.

Verification

Resolved.

Issue G: RelayHub Makes Unguarded Call to Untrusted Contract

Location

https://github.com/LeastAuthority/gsn/blob/3bc59af87bed764cedb828dedbb3a18188b21b0a/contracts/RelayHub.sol#L125

Synopsis

This issue is another vector for the attack described in both Issue C and Issue F with malicious Paymaster contracts. In this instance, the Paymaster burns gas inside the `getGasLimits` method, which is called prior to the `RelayHub` checking any gas limits

Impact

Combined with a malicious Paymaster (Issue C), the balance of a relay (or all relays) can be rapidly burned.

Preconditions

The attacker would need a malicious Paymaster contract and enough balance to appear capable of covering the max cost.

Feasibility

Straightforward, see Issue C.

Technical Details

The attack works in a similar way to the attack in Issue C, with the exception that the gas is burned in `Paymaster.getGasLimits` instead of in `Paymaster.preRelayedCall`. It's worth noting that this call is made before `vars.gasBeforeInner` is set and, as a result, the Paymaster is never charged for the cost of `getGasLimits`. Thus, an on-chain loop and revert inside `getGasLimits` will always cost the relay server.

The `IPaymaster` method `getGasLimits` is defined as a `view` function, this means it can read but not write memory. This code calling this is compiled to a `CALLSTATIC` evm opcode, which will cause a revert if the method tries to write to memory - but it can still read memory, so it can check a mode variable and behave differently.

Remediation

Instead of calling the Paymaster contract to get the gas limits, require that the Paymaster pre-register its `gasLimits` with the `RelayHub`. While this adds one more step to deploying a Paymaster, it will also mean that the `GasLimits` cannot change unexpectedly.

It is worth noting that Solidity also has a "pure" directive, which forbids the method from reading memory. This would appear to fix this issue. However, while Solidity would produce a compilation error if a function declared as pure attempts to read memory, `CALLSTATIC` does not enforce this. As a result, calling an unknown Paymaster must be a view, as opposed to a pure function.

Status

The suggested remediation has been implemented and `getGasLimits` is now called with a gas allowance. This significantly reduces the impact that this attack could have and it is no longer considered a serious security vulnerability.

Verification

Resolved.

Suggestions

Suggestion 1: Improve Documentation

Location

https://docs.opengsn.org/learn/index.html

Synopsis

The interface contracts are well commented in some files, however, coverage is not comprehensive. For example, the Paymaster interface is well commented while the Penalizer interface has no comments present. In addition, a few files have limited descriptions of functionality, design rational, and attack mitigation strategy.

Protocol documentation is provided, however, it does not include the discussion of possible attacks to the GSN network provided in the internal documentation provided to us by the GSN team. While these possible attacks do have some form of mitigation present in the code at this stage, including these issues

in the main project documentation will help increase knowledge of known issues and the rationale behind the decisions made in the protocol design.

Furthermore, documentation defining the expected use cases would be a helpful guide, given that the GSN codebase largely comprises a framework for handling meta transactions.

Mitigation

Include thorough comments for the Solidity contracts describing in detail the intended functionality and expected behavior.

Expand the protocol documentation to include all of documents and comment notes on attack vectors, in addition to currently implemented and possible future implementations of mitigations for various potential attacks.

Finally, we recommend creating documentation that describes the expected use cases and how the system handles requests from beginning to end.

Status

The GSN team has documented the protocol, which has been added to a repository within the GitHub organization. The GSN team has also responded that they have made improvements to the existing developer documentation, which is a continuous, ongoing effort.

Although we also recommend that the GSN team include comprehensive coverage of comments in the code base, the GSN team has said they do not intend to do so and their position is that documentation of code should remain outside of the code.

Verification

Partially Resolved.

Suggestion 2: Consider Implementing an Approve / Confirm Owner Transfer

Location

Any contract inheriting OpenZepplin Ownable contract.

Synopsis

`Ownable.sol` contracts have been a standard in Ethereum, however, there is a minor flaw in their design. The transfer of ownership is vulnerable to accidental updates to mistaken accounts. If any mistaken address is supplied to the update function, it only requires a single transaction to permanently lose ownership of any system inheriting that ownership.

Mitigation

Consider taking a different approach to ownership update by creating two functions for updating ownership. An `approve` function will stage an address in the contract that is intended to become the next owner. If this address is valid then the key (or keys for multisig) holder(s) will then be able to call an `accept` function that will permanently update the ownership to the new address.

This increases the complexity, however, given how infrequent ownership updating is and how important it is to do so correctly, the cost is justified. This also removes the ability to “burn” ownership since one can not create an `accept` transaction from an uncontrollable account. It is not good practice to use a bug as

a feature and that a separate functionality should be created to explicitly burn ownership of a system if this is desired.

Status

The GSN team has responded that they do not intend to mandate how ownership of the Paymaster contracts is implemented. The implementation of ownership mechanisms will be at the discretion of individual organizations that will deploy them as they are outside of the core GSN contracts.

Verification

Resolved.

Suggestion 3: Complete Paymaster Gas Calculation

Location

https://github.com/LeastAuthority/gsn-paymasters/blob/master/contracts/ProxyDeployingPaymaster.sol#L64

Synopsis

A constant declaration `PRE_RELAYED_CALL_GAS_LIMIT_OVERRIDE` for a gas limit is accompanied by a comment with a TODO that the constant should be calculated. This constant appears to increase the gas limit that a Paymaster is expected to need in order to verify transactions from relayers. The value is set to two million gas which is overriding the 210,000 gas that a standard Paymaster uses as a default setting. This increase will expose the relayer to potentially more risk (related to Issue C) and should be considered thoroughly when suggested as a limit.

Mitigation

Provide calculations and rationale for the increased computation, which results in more risk exposure for relayers.

Status

The GSN team has responded that the `ProxyDeployingPaymaster` instance is a special case, which is meant to be trusted and that the security implications of raising the gas limit have been considered in this specific instance. In addition, the GSN team has responded that they are working on a new version, which will use EIP-1167 static proxies, which will no longer require a higher gas limit. However, it remains partially unresolved at the time of this verification.

Verification

Partially Resolved.

Recommendations

We recommend that the *Suggestions* stated above are reconsidered and, if addressed, followed up with verification by the auditing team.

Also, we recommend that the GSN team incorporate the informal documentation and notes provided to our team on potential attack vectors and possible mitigations into the formal protocol documentation. This will increase public awareness of both known vulnerabilities and potential attacks, in addition to suggested mitigation strategies, along with further clarifying the rationale behind the design decisions made by the GSN Team.

Despite the commendable efforts of the GSN team in mitigating the various risks raised in this report, two of the issues present significant challenges that require further diligence. For Issue D, we suggest a further relay girefing attack analysis on the cost of the fee as compared to the opportunity cost griefed on the relayers, and how these costs translate over time with fluctuating value of the fee / gas prices. For Issue E, we encourage the GSN team to continue researching the incentive for relay penalization. If an infraction is not likely to happen frequently, and most nodes are found to have turned off the code that altruistically searches for infractions, there could be no real incentive for any node to report issues under a rational assumption that it costs more to look for infractions without a high likelihood of reward.

Finally, we commend the GSN team for being thorough and rigorous in their approach to security throughout the project. It is clear that considerable time and effort has been applied towards designing and implementing a system which places a strong emphasis on security.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create

an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.