Least Authority

PRIVACY MATTERS

EIP-3074

Security Audit Report

Ethereum Foundation

Final Audit Report: 14 June 2021

Table of Contents

Overview

Background

Ethereum Foundation has requested that Least Authority perform a security audit of the EIP-3074: AUTH and AUTHCALL opcodes specification, which aims to allow Externally Owned Accounts (EOAs) to delegate control of their account to a smart contract. This EIP introduces two Ethereum Virtual Machine (EVM) instructions, `AUTH` and `AUTHCALL`. The first sets a context variable `authorized` based on an ECDSA signature and the second sends a call as the `authorized`, which essentially delegates control of the EOA to a smart contract.

Project Dates

- **May 10 - May 28**: Specification Review *(Completed)*
- **June 3**: Delivery of Initial Audit Report *(Completed)*
- **June 9**: Delivery of Updated Initial Audit Report *(Completed)*
- **June 14:** Delivery of Final Audit Report *(Completed)*

Review Team

- David Braun, Security Researcher and Engineer
- Nathan Ginnever, Security Researcher and Engineer
- Rai Yang, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the EIP-3074 followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following document is considered in-scope for the review:
- EIP-3074: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-3074.md

Areas of Focus + Concern

In Scope

The following are areas of concern have been investigated during the audit, along with any similar potential issues:
- **Wallet Integration:** Signing a single malicious `AUTH` message is sufficient for an adversary to gain control over an EOA. It should be determined if a wallet can be reasonably expected to protect users from this occurrence.
- **Potential Invoker Bugs:** EIP-3074 strongly recommends users interact only with trusted invokers that have undergone rigorous audits and static analysis. However, it is still possible for a bug to be discovered in these trusted smart contracts. This would mean that any EOA that has interacted with the invoker in the past could be at risk. The possibility and range of severities for these bugs in heavily audited invoker smart contracts should be discussed.
- **Existing Smart Contracts:** Currently, it is possible to determine if a smart contract is executing in the first frame of a transaction by checking that `CALLER == ORIGIN`. EIP-3074 allows the transaction's originator to set itself as the `CALLER` at any call depth, therefore breaking the aforementioned assumption. It should be determined how contracts use the above check and, if

existing contracts rely on this assumption, how they would break when the assumption is invalidated.
- **Sponsor-Sponsee Relationship:** This relationship is suggested in EIP-3074, but not thoroughly discussed because it is orthogonal to the actual specification. However, the design of EIP-3074 is highly motivated by the relationship between the sponsor and the sponsee. As a result, an audit of EIP-3074 should examine the interaction between the two parties. Specifically, it should be possible, using primitives provided by EIP-3074, to build a system enabling sponsored transactions, while preventing the following situations:
    - The sponsor receives payment from the sponsee, but the sponsee's intended action is not executed;
    - The sponsee is able to perform an action, but does not reimburse the sponsor; and
    - The sponsee is able to repeatedly waste sponsor resources (e.g. gas fees, nonces, etc.), without incurring a proportional cost itself.
- Anything else as identified during the initial analysis phase.

Out of Scope

The following are areas of concern have not been investigated during the audit:
- Analysis of the safety of the network itself (e.g. Denial of Service attacks); and
- Economic modeling of sponsored transactions.

Findings

EIP-3074 Overview

EIP-3074 offers a flexible primitive to develop novel wallet extensions and introduces a mechanism in the EVM that can make subcalls as an account recovered from an ECDSA signature over the message `keccak256(MAGIC || paddedInvokerAddress || commit)`. This mechanism has widespread use cases, including, but not limited to, sponsored meta-transactions and batched transactions. However, EIP-3074 alters some of the existing security properties of the network and increases the surface area for potential unintended consequences that may occur for both users and existing applications. Given that `CALLER` is the de facto authentication mechanism on-chain, a mechanism that allows smart contracts to masquerade as EOAs, it may introduce new threats and security vulnerabilities.

The goal of our security audit was to uncover and investigate these potential threats and vulnerabilities. Furthermore, our team thoroughly reviewed and examined EIP-3074 and explored all potential security concerns discovered, with a primary focus on the security of user assets, new threat models for users and their wallets, and the feasibility of writing secure invoker smart contracts.

Current Breaking Changes

Tx.origin as the Signer

The `tx.origin` as the signer modification allows for self-sponsored batched transactions for powerful new gas saving techniques and improved user experience. However, this introduces a breaking change first identified in EIP-3074. Currently, it is possible to assume a smart contract is executing in the first frame of execution of the EVM by checking that `msg.sender == tx.origin`. AUTH allows signatures to be signed by `tx.origin`, which allows any subsequent AUTHCALL to set `msg.sender == tx.origin` for the next frame of execution. This is only true for the first frame of execution created by the `AUTHCALL`, but still allows for breaking the aforementioned assumption. Allowing `msg.sender` to equal `tx.origin` is in a class of use cases that the Quilt team calls self sponsoring. This use case is additive to the EIP-3074, and by not allowing for msg.sender to equal `tx.origin` there would be no breaking changes to currently deployed contracts. Most of the benefits of EIP-3074 would still persist given this minimal restriction as well. However, it is necessary to examine its impact on the state of Ethereum to determine if it is safe to proceed with allowing the self sponsoring use case.

The Quilt team has identified a currently deployed use case that relies on checking that `msg.sender == tx.origin`:

- Naive flash loan entry rejection: It is possible to prevent flash loans from calling a smart contract's functions by ensuring that `tx.origin == msg.sender`. This would imply that only EOAs can call the functions.

We estimate the security impact of these known breaking changes to be minimal. While these use cases are valid and do occur, the Quilt team has commissioned an independent third-party team, Dedaub, to scan the Ethereum state for any smart contracts that appear to be using this check. This investigation was conducted in parallel to our review of EIP-3074.

There needs to be confidence that all occurrences of `tx.origin` for frame execution will not lead to critical security issues. Furthermore, the mitigation actions and coordination efforts required to update and fix all known existing smart contracts using `tx.origin` was being determined at the time of this security review. We expect the effort required to address the `tx.origin` concerns will be reasonable.

The published findings from the Dedaub scan seem to agree with our analysis. We quote from the findings conclusion: “As a result, we rate the impact of EIP-3074 as ‘moderate but manageable’, yet acknowledge that the results are subject to interpretation.” With Suggestion 4 we encourage an increased awareness of the issues surrounding the use of `tx.origin` as a flash loan prevention.

EOA Detection with `msg.sender == tx.origin` Blocks Sponsored Transactions

EIP-3074 supports sponsored transactions where subcall’s `tx.origin` is sponsor and `msg.sender` is the sponsee. However, the current contract which uses naive flash loan protection with `msg.sender == tx.origin` would block the sponsored transaction. We expect this concern needs to be raised and a mitigation should be developed in the same way as self sponsored transactions.

Unknown Breaking Changes

While we have only explored and discussed the known breaking change centered around access control with `tx.origin`, there may be other breaking changes stemming from allowing a smart contract to forward messages as a sender equal to the origin. However, we have not been able to identify any issues resulting from this.

We do not anticipate there being many unknown uses of `tx.origin` that will break simply. Warnings in developer threads about the unsafe assumption that there is a difference between an EOA and a smart contract exist, and it is considered possible, by the developer community, that the relationship between smart contracts and EOAs will change in the future. As a result, developers have been reluctant to use `tx.origin` due to uncertainty about its future. The adoption of Account Abstraction (AA) will further minimize the difference between EOAs and smart contracts. As such, we suggest that developers avoid using `tx.origin` because this relationship will likely change further with AA.

In summary, we suggest continued efforts to move development away from the use of `tx.origin` as a security measure.

Signed Data Security

Type (“Magic”) Prefix

The type byte is enforced in the protocol and must be present in the signature. There has been a debate in the Ethereum developer community about the security validity of using a prefix to separate Ethereum transactions from those signed by an account for other purposes. We find that ensuring EIP-3074 signatures do not collide with the signature space of regular Ethereum transactions is a necessary security feature. The type byte for EIP-3074 will be `0x03`, which will set their signatures in their own domain entirely.

Invoker Domain Separation

The invoker address is in the signature in the protocol and creates a base layer of domain separation security on which invokers signatures should work. This is functionally equivalent to the `address verifyingContract` field of the EIP-712 domain separator. This domain separator is particularly important as it provides an immutable pointer to the verification logic expected. If this address points to a malicious invoker, the signing account will be giving full access to its contract calls to an attacker via the logic that the malicious address points to. These addresses are the last line of defense against a malicious invoker and the only area that needs to be attacked for a successful full breach of an account.

Address Manipulation

We highlight the importance of allowlisting invoker addresses and note that the executable code these addresses point to must be immutable. Care should be taken that wallets allow EIP-3074 signatures only when a correct address is in the transaction being signed and the executing code is not upgradeable or deployed with CREATE2.

A recent study was done on cognitive bias as a weapon to attack Ethereum users that could provide some more insights into attack vectors involving correct usage of the invoker address. The paper introduces two classes of social engineering attacks called “homograph” and “address manipulation” and claims the introduction of six possible attacks from these classes. We found that the attacks here share a similar threat model as malicious invokers and the homograph, or falsification of typographic symbols, is interesting for this review as wallets will need to take considerable care when implementing an allowlist of these invoker domains. Generally, we believe that social engineering or text manipulations can be handled in the wallet’s allowlist and most of the attacks in the research do not pertain specifically to invokers.

Wallets should take measures to prevent the signatures of unknown invokers, considering social engineering attacks or any possible manipulations of an address that may make their way through the allowlist before reaching the signing API. A hardcoded address in the wallet implementations will help remove the threat of social engineering attacks.

Incomplete Fields Restriction in Commit

In the signature, the commit field can include `nonce`, `chainID`, `to`, `gas`, `value`, `valueExt`, `calldata`, `expiry`, and others. Any fields missing from the commit, or included but incorrectly validated, are susceptible to external manipulation. For example, if `calldata` and `to` are not included, an attacker can make an invoker `AUTHCALL` a random call to any smart contract on any function.

In positive cases, the incomplete fields provide useful functions such as a social recovery feature similar to a multisig smart contract wallet whereby a user who lost a private key may have an authorized trusted third party (friend) run transactions on behalf of the user. In this case, only the friend’s public keys are included in the commit. The user must trust the friend not to manipulate the transaction pre-image (`to`, `gas`, `value`, `valueExt`, `calldata, etc`). Another example of a positive use case is the `alarm clock transaction(empty commit)` which can be validated by having `calldata` hardcoded in the invoker.

Replay Protection

The current data signing specification does not enforce any other domain separation fields beyond the type byte and invoker address. The signing specification does not enforce a `nonce`, `chainID`, protocol salt, or version number. Signing these commit data fields provides application level verification that can be used for various replay protections that will need to be considered for each invoker created. The application specific replay protection can, but is not required to, be baked into the `commit` to give flexibility for different use cases and save gas.

Any set of fields used to verify transactions in the invoker should guarantee that all security critical replays are impossible. This is a requirement and necessity for every invoker that requires replay protection. We acknowledge that there are potential uses of invokers that may not require a nonce or any replay protection. The EIP-3074 protocol enforcing any fields beyond the verifier contract address would lead to transactions wasting data and gas when not needed. We believe leaving the commit data contents up to each invoker's use case is a reasonable approach that offers the most flexibility.

Human Readability

The `commit` value presents a security concern for signing transactions if not implemented correctly because it is not human-readable. This is also true for all hashed Ethereum transactions. It should be required that EIP-3074 compliant wallets use a standard way of interpreting inputs of the hash and the subsequent signing of the commit hash for each invoker.

One approach to improving the security of signatures is typed structure data hashing and signing (EIP-712), which gives the user greater transparency into what is being signed than a single hash value. However, we do not think the added complexity of EIP-712 is necessary for this proposal because it is expected that signatures will be made only for invoker smart contracts chosen from an allowlist of trusted smart contracts.

Users can be expected to trust wallets using an EIP-3074 standard interpretation of signed data and that the wallet will verify the address field to make sure that only calls to safe invoker smart contracts are allowed. The wallets will also need to ensure that the call parameters for the invoker contracts are being formatted and signed correctly for each supported type of invoker.

Invoker Implementation Security

Potential Malicious or Incorrect implementations (Non-Exhaustive)

The following are two potential security issues, but these are not exhaustive of what is possible:

- Malicious Trusted Proxy Invoker: An invoker that does not validate any commit data in the signature, authenticates with signature, and sends any malicious call.

  This version of a trusted proxy demonstrates why it is critical to have restrictions such as replay protection in the invoker contract and an allowlist of trusted and properly implemented and audited invokers only used in wallets. An example of a trusted proxy invoker is self-sponsoring transactions. The required verification is that the AUTH call that sets the authorized context address is the same address as the sender of the transaction. This is likely the most important verification for trusted invokers and can be done with simply signing the self-sponsored address in the commit data and ensuring that `msg.sender` is supplied as the commit data to AUTH. An example from the Quilt team shows how to achieve this verification in the commit that is signed by the user of the invoker:

  ```
  bytes32 commit = bytes32(uint256(uint160(msg.sender)));
  ```

However, nonce replay protection is still advised to prevent signed payloads from being reused unexpectedly by the self-sponsored user.

- Incorrectly implemented invoker:
    - Calldata is not validated in the invoker and checked to be the same `calldata` that is signed in the commit;
    - There is no replay protection by nonce (with the exception for `alarm clock` transaction);
    - Incomplete commit fields validation in addition to nonce include, but are not limited to, `to`, `gas`, `chainID`, `calldata`, `value`, and `expiry` (this list is non-exhaustive);
        - In the case of replay protection by nonce only, the attacker steals the signature and manipulates other fields to get the invoker to send malicious transactions. For example, a missing `chainID` verification can let an authorization exist on side chains implementing the invoker.
    - Transaction order ignored in a batch transaction such as approve/transfer, where the second transaction depends on the status of first transaction;
    - No limits for the number of transactions in a batch transaction;
    - No limits of gas in transactions;
    - No sponsor/sponsee relation management; and
    - Invoker upgrades to a malicious invoker, which can be mitigated by not allowing an invoker address created by CREATE2.

Permanent Authorization Concerns

Amplified Attack Consequences

If an invoker does not code the correct usage of replay protection in the commit data, then an invoker may become permanently authorized to use the account in any way it would like. If the invoker is malicious or incorrectly implemented, attacks will be amplified by permanently authorized transactions, especially compared to current meta-transaction smart contract implementations. The attacker could drain all of the user's non-Ether assets by sending an arbitrary number of transactions to the invoker smart contract or in batches. This amplification attack is made possible by the ability to create multiple transactions without the need to be careful about how the attack is deployed given that an invoker could simply proxy any call. It is worth noting there are currently meta-transactions that could have malicious smart contracts as their receiver that can drain smart contracts holding multiple assets of value pooled together from multiple accounts.

Permanent authorization changes the assumptions of transactions. It can currently be assumed to be true that malicious signatures will only be valid for one transaction broadcast to the network at a fixed point in time. After EIP-3074 is introduced, a single signature may be able to allow multiple transactions extending into the future with that signature. This adds an extra layer of concern that must be examined when trusting an invoker. The possibility for invokers to become malicious in the future must be considered given that authorization is permanent.

This amplification is limited in nature, and requires that an invoker smart contract is implemented incorrectly, which is completely avoidable. This amplification is also no worse than unlocking a geth account and allowing a corrupted wallet to have free access over signatures on any given single account in the compromised wallet. Given the attack surface area is no worse than previous wallet concerns in Ethereum, we do not see this amplification as a blocker to implementing EIP-3074.

Wallet Implementation Security

Malicious or Unintended Signature

A malicious signature, unintended signature due to a corrupted wallet, or human error could cost the user non-Ether assets, even in the case of a perfect invoker smart contract. Given the nature of security responsibility from the wallets implementing the EIP-3074 invoker signatures correctly, we express our concerns with the wallets. To reduce the risk of a malicious or unintended signature, we encourage:

- Users to use a safe and vetted wallet (e.g. Metamask);
- Wallets to show a warning when users are signing an EIP-3074 signature; and
- Invoker smart contracts set the limit of the number of transactions in a batch transaction.

Hardware Wallet Security

Hardware wallet signing to malicious invokers can be mitigated by only allowing signing to allowlisted invokers and upgrading the firmware to update the allowlist.

Hardware wallet signing unintended signatures due to the lack of a human readable message can also be improved in the future by showing more signed messages in the wallet UI.

Inconsistencies of Signed Data API Implementations

Metamask provides an overview of the history of implementation issues of signed personal data. It is generally recognized that creating an EIP or getting consensus is desired before implementing some new API's, particularly those dealing with signatures. This provides consistency between wallets preventing the APIs from diverging where an application using one wallet with an API method under the same namespace as another wallet has different implementations. This has happened with signed personal data as a few wallets decided to implement the signature API before consensus was reached or before the EIP was finalized leading to multiple versions of the signing API.

EIP-3074 will require a new signing API and we suggest that the signing fields are not changed before wallets begin implementing EIP-3074 signatures (Suggestion 2). The Quilt team has replied that given how simple the signing implementation is, and that there are no current changes planned, that this should not be an issue. We simply acknowledge past experience in this area for visibility. The Quilt team has a fork of the Go-Ethereum wallet with an API method that allows for EIP-3074 signatures. We suggest rallying the other wallet providers around this method for consistency.

Allowlisting Invoker Smart Contracts

From a security perspective, the most important implementation detail for wallets will be the allowlist. Resources must be provided to help users understand the importance of the allowlist in the invoker smart contract, and to provide a standard path for implementation (Suggestion 5).

Sponsor-Sponsee Relationship

The Sponsor-Sponsee relationship and interactions can be managed by the invoker. In the case of an EIP-3074 invoker, the sponsor must pay for the Ether gas costs. However, there is the possibility to use an invoker that can withdraw an ERC-20 token from the sponsee as payment equivalent for the gas costs, exposing EIP-3074 to most of the concerns faced by gas payment alternatives like GSN. Creating an open relayer network out of an EIP-3074 invoker will introduce complexities associated with distributed networks. Some of the issues and necessary mitigations that may arise include:

- The sponsor receives payment from the sponsee, but the sponsee's intended action is not executed;

    - This can be mitigated by an upfront sponsee deposit to the invoker smart contract and the invoker smart contract only sends the payment to the sponsor when the action is successfully executed.
    - In order to prevent the sponsor from overcharging the sponsee on `AUTHCALL` failure, `AUTHCALL` should be able to distinguish two types of faults:

      sponsor-attributable fault:

        - `Gas_left < subcall_gas` where a sponsor does not pay enough gas for `AUTHCALL` under the assumption that the sponsor is responsible for validation and submission of subcalls with a reasonable gas limit.
        - Invoker's Eth balance < `subcall_value.`

      For sponsor-attributable fault, EIP-3074 specifies that `AUTHCALL` should exit to the parent frame and all deposited funds be refunded to the sponsee, and the sponsor would be charged for all gas used/available. Therefore the sponsor is charged for their fault and will not be able to overcharge the sponsee more gas than `gas_left`.

      sponsee-attributable fault:

        - Subcall fails on out of gas or another error.

      For a sponsee-attributable fault, `AUTHCALL` would return `0` and the invoker catches the fault and does not refund the sponsee the gas consumed by the subcall. In this way, the sponsee is charged for faults attributable to it as well.

- The sponsee is able to perform an action, but does not reimburse the sponsor;
    - Mitigated the same as above. An invoker would only AUTHCALL the subcall if there are sufficient funds deposited by the sponsee.
- The sponsee is able to repeatedly waste sponsor resources (e.g. gas fees, nonces, etc.), without incurring a proportional cost itself. If the sponsee has to pay the cost upfront as above, therefore incurring cost, otherwise potential issues and mitigations are:
    - If the sponsee repeatedly delegates many transactions to the sponsor causing waste/draining of gas, the mitigation can be rate limiting the sponsoring requests from sponsee by the sponsor's API, or by requiring fees be paid to the sponsor in any token.
    - If the sponsee puts too many transactions in one batch, wasting/draining gas from the sponsor, a mitigation can be for the invoker to set a gas limit per transaction and a limit on the number of transactions per batch by the invoker.
    - If the sponsee maliciously puts many invalid transactions into one batch, wasting/draining gas, this can be mitigated by setting the number of transactions per batch limit by the invoker, along with proper error mitigation in a pull style batch rather than pushing an array of transactions with potential errors that could block other users in the batch.
    - When `value!=0`, the sponsee must fund the invoker for the sponsored transaction up front, or the sponsored transaction will not go through.

In general, we consider using EIP-3074 to be a tool that can help define the role of relayers in a network like GSN. For example, it may be possible to reduce the complexity of the `RelayHub`, `PayMaster`, and `TrustedForwarder` contracts of the GSN network into a single invoker contract that passes `msg.sender` to the Recipient Contract for certain use cases. EIP-3074 can replace the `TrustedForwarder` and standardize a few minimal use cases while still making use of the GSN network of relayers. This reduction in state and complexity will lead to less costs and increased security.

However we note that GSN is complex because it aims to be flexible enough to fit many use cases. This leads to carefully planned contracts enforcing trust rules to mediate a sponsor-sponsee relationship in terms of gas payment for all use cases. EIP-3074 will be expected to encounter the same set of trust requirements being enforced in the invoker contract which may end up making this relationship infeasible to produce safely for all use cases.

Sponsor-sponsee invokers attempting to serve open-ended use cases will need to handle fair gas usages in ways that GSN has created. For example, a side-chain bridge may set a simple rate limiter to mitigate abuses of a subsidy meant to make bridging easier for users, however, this will be inherently vulnerable to the side-chain spamming the bridges with transfers to the more expensive chain, forcing the bridge to disable sponsoring fees in the expensive direction.

We expect some minimal use cases to be reasonable where requirements on gas spending can be relaxed. For example, a dApp can allowlist a trusted set of users in the invoker to spend the funds in a `PayMaster` contract for siloed dApps.

Future Security Concerns

Source of Value: ValueExt Potential Change

According to EIP-3074, it is not possible to deduct Eth from the authorized account. Since the ability to do so is desirable, a future provision was made by introducing the (currently unused) `valueExt` operand of `AUTHCALL`. As noted in the EIP, there are some issues to be sorted out regarding the impact of its use on the transaction pool. Handling these issues is non-trivial, and it is therefore our assessment that it is prudent to disable this feature in the current proposal (while reserving the operand to avoid the need to introduce a future opcode).

We do not see any security concerns with the current implementation of `valueExt`. A potential concern when it is enabled in the future is that there is currently only one failing condition for a transaction in the mempool which is that the nonce has already been included in a block. Introducing both `valueExt` or AA will create a new class of mempool errors where a transaction may invalidate another based on the balance of Eth in the account. This added failure case could cause the application expecting transactions to succeed to need refactoring.

General Security Conclusions

Signatures have been a point of security concern since Bitcoin and the prefix `'\x18Bitcoin Signed Message:\n'` which first introduced the minimal protocol for separating bitcoin transactions from signed data. It is strongly suggested that the first byte of a signed transaction payload be a type byte. The danger is accidental or malicious signatures being compatible for multiple transaction types. This is the first line of defense against transaction replay attacks. Transaction signing is a well studied problem and wallets already have been dealing with full access security. Meta-transactions have as much access as there is in the possible space of corrupt smart contracts and have been implemented and used in high profile projects under well audited conditions. We feel that EIP-3074 and invokers are in the same category of concern. The only potentially amplified concern is malicious access to all of the signing account's capabilities over an extended period of time. This sets EIP-3074 into a slightly more concerning security space than meta-transactions, but not much more than a properly implemented wallet. We conclude that under the right conditions - wallets and invokers being implemented correctly - that this proposal is safe for use.

Allowlisting capability may improve the security of currently deployed meta-transactions. In the current state of meta-transactions, all dApps must create their own set of meta-transaction receiver smart contracts, and there have not been many standards around standard meta-transaction use cases for developers to reuse. In the ERC-20 contract ecosystem for example, we see a standard set of contracts that most teams choose to use, namely OpenZeppelin, which reduces the chances for incorrect implementations. However it still leaves room for incorrect deployment or usage.

A feature of EIP-3074 that minimally increases security over meta-transactions is the ability to reuse invoker smart contracts across dApp platforms. EIP-3074 invoker smart contracts can function like libraries that the community can agree upon and deploy only once which could reduce chance occurrences of incorrect deployment. This feature may introduce concerns about single points of failure. The consequences of an incorrectly implemented invoker could be more widespread if all dApps now use the same invoker. The ability to deploy once gives wallets the ability to allowlist known and acceptable invokers, and assuming that a perfect invoker can be created, the chances for mis-signing a corrupt invoker can be completely eliminated.

The allowlist must be strictly enforced on EIP-3074 transactions and we suggest that invokers are simple in complexity to increase the probability that no mistakes occur in the invoker specifically. This task does not seem too difficult for wallets to execute if they can effectively work together.

There are a few areas of security worth noting in which invoker smart contracts can reduce risk for users. To avoid multiple approval transactions many dApps will ask users to approve an infinite amount of tokens to the application. A cheaper or more user friendly way to approve and transfer will help reduce the use of this pattern. The property that invoker smart contracts can be made to service multiple applications makes this proposal appealing in terms of reducing the chances for incorrect or malicious code being introduced to the users by focusing on single implementations.

Another ERC-20 example is tokens using Permit meta-transactions. Each token has to ensure that the approvals are implemented correctly in their independent code bases. This same functionality may be achieved with one well known and reused invoker that batches any ERC-20 approve/transfer.

Allowing code to execute on behalf of an EOA must be done carefully, however we do not see any reason why some use cases of EIP-3074 can not be implemented securely.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, 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
Suggestion 1: Create Invoker Smart Contract Prototypes	Reported
Suggestion 2: Ensure EIP Does Not Change for Standard Signature Implementation	Reported
Suggestion 3: Create EIP for Standard Invokers	Reported
Suggestion 4: Create More Awareness of Flash Loans and tx.origin Access Controls	Reported
Suggestion 5: Provide Wallets With Any Necessary Allowlisting Guidance	Reported
Suggestion 6: Promote Simple Invokers	Reported

Suggestion 7: Test Opcodes Implementation Thoroughly	Reported

Suggestions

Suggestion 1: Create Invoker Smart Contract Prototypes

Synopsis

EIP-3074 is admirable in its simplicity but raises many “what-if scenario” type questions. Building prototypes is a time-honored practice in software development to lower risk and to test basic assumptions. The Puxi (浦西) Testnet is a good starting point for experimentation with the new opcodes and should be expanded to include sample smart contracts similar to AuthorizeProxy.sol and EIP3074Relayer.sol.

Mitigation

We suggest that several fully-functional example invoker smart contracts that exercise the `AUTH` and `AUTHCALL` opcodes be written for the implementation of anticipated use cases (sponsored transactions, batched transactions, etc). The smart contracts need not be secure (because prototypes should be inexpensive) and it should be made clear that they are not to be considered reference implementations. The smart contracts should include automated tests. Such smart contracts would provide a basis for learning about the proposal, experimentation, and further conversations regarding anticipated scenarios. They would make it possible to try out various adversarial attacks and to exercise edge cases.

Status

Reported.

Suggestion 2: Ensure EIP Does Not Change for Standard Signature Implementation

Synopsis

There has been a history of signature API issues that can be avoided when creating a new signing API for EIP-3074. The primary issue is ensuring that EIP-3074 does not change after wallets choose to start implementing it. If it does change there could be another situation where one wallet has a v0 implementation of the API while another has v1, the more modern implementation reflecting changes in the EIP-3074.

Mitigation

We suggest EIP-3074 does not change after the signing API is implemented.

Status

Reported

Suggestion 3: Create EIP for Standard Invokers

Synopsis

The reusability of invoker smart contracts in some use cases will allow for standard invokers to be produced. dApp smart contracts will not necessarily have to be programmed specifically with code to handle meta-transactions or to find insecure ways of reducing gas costs (e.g. infinite ERC-20 approvals).

Removing this necessity that each dApp is aware of the invoker allows for some invokers to service multiple dApps.

Mitigation
We suggest using a public forum where consensus on safe, well audited, and vetted invoker smart contracts can be discussed like the EIP system.

Status
Reported.

Suggestion 4: Create More Awareness of Flash Loans and `tx.origin` Access Controls

Synopsis
We speculate that access control is the main use case that will contain any breaking changes after EIP-3074 introduction. Research by the Dedaub team has been conducted to identify all known smart contracts that will have a vulnerability introduced after the acceptance of EIP-3074. This research is targeted mostly at checks for `tx.origin == msg.sender`. However the research is not limited to this check.

Mitigation
Publish research that shows a high degree of confidence that all currently deployed smart contracts that will have broken security assumptions after EIP-3074 can be reasonably mitigated. This research will help remove some of the concerns of the breaking changes to currently deployed smart contracts that utilize `tx.origin`.

Available research suggests that the breaking changes will have an impact that is manageable for currently deployed contracts. We suggest that available research be publicized with emphasis placed on the issues surrounding flash loans, sandwich attacks, MEV and the usage of `tx.origin` as an insufficient safety measure.

Status
Reported.

Suggestion 5: Provide Wallets With Any Necessary Allowlisting Guidance

Synopsis
If invoker smart contracts can be reused then it is not unreasonable to require wallets to agree on what invokers they will support and block all other invokers from being signed in their wallets. This will reduce all of the known attack vectors so long as the invoker smart contracts are well audited.

Mitigation
Given how important the allowlist is, providing a standard method for implementing, and updating the allowlist for all wallets could be beneficial for security. For example, only invoker addresses that have been cleared through a community consensus process like the EIP should be considered for allowlisting. After being considered, all wallets should agree to add the address to their allowlist in a timely manner.

A wallet which supports the signature with a specific commit should have a corresponding allowlisted invoker.

The invoker address created by CREATE2 should not be trusted to avoid malicious invoker upgrades to the same address.

Status

Reported.

Suggestion 6: Promote Simple Invokers

Synopsis

Given that invokers will likely be allowlisted and used by everyone, there must be no mistakes in their implementation. The consequences of a mistake are spread throughout all that have interacted with the invoker. Compared to a subset of contracts that have a bug in them that can be exploited, a single invoker could reach more accounts.

Mitigation

We suggest building and promoting invokers that are simple in logical complexity. If the complexity of an invoker is beyond what formal verification can capture, the system may have logical complexities that are missed by code review. We do not have a specific suggestion for what should be considered too complex, but we do understand that a smaller surface area has less chances of containing issues. The ideal situation is full coverage of the attack surface area of any invoker that is in a wallet's allowlist for all accounts to use.

Status

Reported.

Suggestion 7: Test Opcodes Implementation Thoroughly

Synopsis

The opcodes AUTH and AUTHCALL are implemented in a fork of Ethereum client Geth in Puxi testnet, it needs to be tested thoroughly before it's merged in the next fork of the mainnet.

Mitigation

We suggest creating unit and integration tests to include all edge cases and scenarios, ensuring the opcode is implemented correctly and safely as specified in the EIP. Sufficient test coverage helps to identify potential edge cases, and helps protect against errors and bugs, which may lead to vulnerabilities or exploits.

Status

Reported.

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.