Dedaub

Security Technology for Smart Contracts

Impact Study of EIP-3074

ethereum foundation

Date: May 19, 2021

Abstract

Dedaub was commissioned by the Ethereum Foundation to perform an audit/study of the impact of Ethereum Improvement Proposal (EIP) 3074 (AUTH and AUTHCALL) on existing contracts.

In order to appraise the impact of the proposed change, we performed extensive queries over the source code and bytecode of deployed contracts, inspected code manually, examined past transactions/balances/approvals, and informally interviewed developers.

Executive Summary

The extent of the impact is not straightforward to ascertain. There are many affected contracts, estimated at 1.85% of unique (i.e., contracts with the same bytecode are counted once) active deployed contracts. Many of these contracts handle substantial sums and interact with an Automated Market Maker (AMM), such as Uniswap or Balancer. The result is that the contract is being threatened with flash-loan or other pool-tilting attacks.

However, our considered opinion after this study is that a) the impact will be significantly limited with appropriate awareness of the upcoming change; b) the vulnerable code patterns are already exploitable by miners and flashbots, and such exploits will become more threatening in the near future. As a result, we believe that the impact of EIP-3074 is certainly manageable and perhaps a net positive in the overall security of the Ethereum blockchain ecosystem. We recommend reading the section titled "Opinion" at the end of this report for more detail and documentation of our (subjective) opinion. The objective, numeric findings of the study are listed in detail in the "Experimental Findings and Study" section.

Setting and Background

The focus of the study is to examine to which extent existing contracts are adversely impacted by the changes introduced by EIP-3074. The most significant impact of EIP-3074 (in its current "strong" form) on *past* contracts is the inability to reliably distinguish the `msg.sender` (in Solidity) of a transaction. In particular EIP-3074 enables programmatically setting the transaction's `msg.sender`, thus rendering obsolete common checking patterns such as "`msg.sender == tx.origin`". This pattern is often used to ensure that a contract's caller is an Externally Owned Account (EOA) and not a smart contract, and, thus, cannot have manipulated on-chain quantities atomically without the rest of the environment (i.e., real-world actors) getting a chance to correct or punish such manipulation. As the study confirms, protection against flash-loans in contracts that interact with AMMs is the primary use of such patterns.

Experimental Findings and Study

Task 1: Source Queries

Retrieve all contracts with msg.sender == tx.origin in their published source.

As the first quick experiment, we queried our database (behind contract-library.com) for contracts with common combination patterns between `tx.origin` and `msg.sender` in their source. This means that there are two sources of *incompleteness* in this query: a contract may not have source, or a contract may be checking `tx.origin` against `msg.sender` but without using this exact source code pattern. On the other hand, this query is over a fairly complete set of contracts, virtually all contracts ever deployed (with very minor exceptions). Notably, the numbers concern accounts, i.e., contracts with the same code are counted as many times as deployed. (This aspect will be different in our next experiments.)

```
select hex(a.address)
from address a
join source_code s on a.md5_bytecode = s.md5_bytecode
where s.code like '%require(msg.sender == tx.origin)%'
and a.network = 'Ethereum';
```

874 rows

```
select hex(a.address)
from address a
join source_code s on a.md5_bytecode = s.md5_bytecode
where s.code like '%msg.sender == tx.origin%'
and a.network = 'Ethereum';
```

2312 rows

Clearly, the number of contracts that employ the pattern is substantial.

Task 2: Bytecode Queries

Retrieve all contracts with msg.sender == tx.origin checks regardless of whether the contract has published source and of whether the contract has this exact code pattern or merely an equivalent one (e.g., getting msg.sender through an internal function, which is common).

In order to reduce the time overhead of our queries and not have our dataset be dominated by old and unused contracts, we limit our dataset to contracts that have transacted recently (within 200K blocks from block number 12374455). This is effectively all contracts that saw activity in the past month. We also disregard duplicates: contracts with the same bytecode are counted only once, regardless of how many times they are deployed. We end up with 34,962 unique contracts. For comparison, the total unique bytecodes in the DB of contract-library.com (up until the same block) are 398,275. By limiting our attention to contracts that transacted in the past month, we reduce our workload by over 10x, allowing more targeted exploration and quick experimentation with the most relevant contracts.

After decompiling the contracts using the gigahorse decompiler, we ran a simple analysis for the detection of comparisons between `msg.sender` and `tx.origin`. Because of the way our query is written (to enable completeness of the results) this will consider both "`msg.sender == tx.origin`" and "`msg.sender != tx.origin`", as well as any other data-flow combination.

Through our pipeline we are able to successfully analyze 99.8% of all contracts in our dataset (with 41 decompilation timeouts) finding that the comparison between `msg.sender` and `tx.origin` is present in **1.85%** of them (**648** unique contracts).

Sanity checking of static analysis completeness:
The analysis detecting the pattern of this task will be used as a building block for the queries of the next tasks. Because of this, it is crucial to evaluate its completeness. To do so we reran the 2nd database query of task 1 (the one that returned 2312 rows), this time limiting its results to contracts transacted in the last 200K blocks. This query of our source database returned 387 unique contracts.

The sanity check is how many of the 387 are not in the 648 flagged by the bytecode-level analysis. The sanity check returned 59 contracts, i.e., the analysis would seem to have 85% recall. We sampled the first few: in most cases the guard is not present in the deployed contract but its deployers had uploaded the source of several files on etherscan and some other file had the guard. We had to sample 8 (of the missing 59) files before we found the first actual false negative for the static analysis. This leads to the conclusion that the 387 number is greatly inflated, therefore the static analysis at the bytecode level misses very few actual combinations of `tx.origin` and `msg.sender`.

Conclusion: the static analysis of bytecode is solid, finding at least ~98% of real guards that combine `msg.sender` and `tx.origin` and twice as many as a source-level query.

We now have a solid base for continuing with deeper analysis combinations. **For the tasks that follow, we report results over the 648 contracts (1.85% of all active contracts of the past 200K blocks) returned by the analysis of this task.**

Task 3: Revert for EOA Caller

Retrieve all contracts that do (effectively) require(msg.sender == tx.origin), i.e., revert (directly or soon thereafter) when the above check pattern fails.

In the next experiment, we examine more closely (yet still with automated analysis) the results of the previous step. We detect instances of the comparisons between `msg.sender` and `tx.origin` produced for Task 2 where the result of the condition flows to a conditional jump that can control whether the program will reach a `REVERT/THROW` statement or not.

Our query is flexible enough to recognize simple guards such as "`require(msg.sender == tx.origin)`" but also more complex patterns such as "`require(isApproved(msg.sender) || msg.sender == tx.origin)`" while being agnostic to the exact shape of the checking code.

The results of our query show that such a pattern is present in **94.44%** (612 unique contracts) of the contracts returned by Task 2.

Task 4: Automatic Classification: AMM Interactions

Retrieve all contracts that have msg.sender == tx.origin that guards an interaction with an AMM (Uniswap, Balancer). This indicates that the pattern is used as flash loan protection.

We next use static analysis to determine the extent to which a `tx.origin+msg.sender` guard pattern clearly protects AMM interactions. **This will necessarily be an under-estimate**! The code semantics could be arbitrarily complex. We capture AMM interactions that are discernible in the same contracts and that can clearly be affected by price manipulation--e.g., swaps.

We provide two static analysis variants:

1. The first, optimized for completeness, detects programs that have at least one instance of the condition that combines `msg.sender` and `tx.origin` and at least one external call with a function signature matching the Uniswap and Balancer APIs we model, even if our analysis cannot detect a way for them to be part of the same transaction execution.
2. For the second variant, we optimize for precision, detecting instances of the conditions that combine `msg.sender` and `tx.origin` produced in Task 2, where the condition *can be followed* by an external call with a signature matching the Uniswap and Balancer APIs we model.

The API calls we consider are the following:

- Uniswap/Sushiswap:
  - `swapExactTokensForTokens(uint256,uint256,address[],address,uint256)`
  - `swapExactTokensForTokensSupportingFeeOnTransferTokens(uint256,uint256,address[],address,uint256)`
  - `swapExactTokensForTokens(uint256,uint256,address[],address,uint256,bool)`
  - `swapTokensForExactTokens(uint256,uint256,address[],address,uint256)`
  - `swapTokensForExactETH(uint256,uint256,address[],address,uint256)`
  - `swapTokensForExactETH(uint256,uint256,address[],address,uint256,bool)`
  - `swapExactTokensForETH(uint256,uint256,address[],address,uint256)`
  - `swapExactTokensForETHSupportingFeeOnTransferTokens(uint256,uint256,address[],address,uint256)`
  - `swapExactETHForTokens(uint256,address[],address,uint256)`
  - `swapExactETHForTokensSupportingFeeOnTransferTokens(uint256,address[],address,uint256)`
  - `swapETHForExactTokens(uint256,address[],address,uint256)`
  - `swapETHForExactTokens(uint256,address[],address,uint256,bool)`
  - `getAmountsOut(uint256,address[])`
  - `getAmountsIn(uint256,address[])`
  - `swap(uint256,uint256,address,bytes)`
- Balancer
  - `swapExactAmountIn(address,uint256,address,uint256,uint256)`
  - `swapExactAmountOut(address,uint256,address,uint256,uint256)`
  - `joinswapExternAmountIn(address tokenIn, uint tokenAmountIn, uint minPoolAmountOut)`

- `exitswapExternAmountOut(address tokenOut, uint tokenAmountOut, uint maxPoolAmountIn)`
- `joinswapPoolAmountOut(address tokenIn, uint poolAmountOut, uint maxAmountIn)`
- `exitswapPoolAmountIn(address tokenOut, uint poolAmountIn, uint minAmountOut)`

The results of our analysis variants are:

1. 131 contracts (20.22% of the 648 contracts of Task 2) are detected by the completeness-optimized variant (merely containing both a Uniswap/Balancer API call and a `tx.origin+msg.sender` guard).
2. 104 contracts (16.05% of the 648 contracts of Task 2) are detected by the precision-optimized variant (containing a Uniswap or Balancer swap that the analysis determines to be controlled by the `msg.sender+tx.origin` comparison).

The discrepancy suggests that the static analysis is incomplete. This is expected. One source of incompleteness is mere program complexity. In contrast to the previous, >98%-complete, analysis, this one is much harder: the statement "`tx.origin == msg.sender`" (or any variation) could be very far in the bytecode from the AMM function call. To estimate the impact of this source of incompleteness we inspected 5 of the contracts among those in the set of 131 but not the set of 104. In one of them the condition is used to control the execution of an AMM call while in the four others the condition is used to disallow accidental ETH receivals from EOAs (but accepting any contract). Therefore the 104 number is more reliable than the 131.

In addition to the above, the analysis produces an underestimate (i.e., incomplete results) for two reasons:

1. There may be AMM APIs that we do not take into account. We believe that we have covered the most frequent cases.
2. Our static analysis cannot reason about more complex inter-contract interactions. We can detect direct interactions with AMMs but not indirect ones (i.e. the analyzed contract calls another contract which interacts with an AMM).

The impact of these sources of incompleteness will become clear with the manual inspection of task 6.

To summarize, we estimate 104-of-648 (16.05% of uses of the pattern, 0.30% of all contracts) to be a lower bound of contracts using the guard for flash-loan/price manipulation protection because of interactions with an AMM.

Task 5: Automatic Classification: Reentrancy Protection

Retrieve all contracts that have msg.sender == tx.origin as a guard over code that performs an external call. Consider which of these calls would correspond to a possible reentrancy pattern, under both strict and loose definitions.

In the next task, we consider whether a `tx.origin+msg.sender` guard may be used for reentrancy protection (or may inadvertently provide it, although this is a much harder question). Invalidating such guarding, via EIP-3074, can render unsuspecting contracts vulnerable.

Although our analysis will be necessarily incomplete, it is good to contrast it with the analysis of the previous task, for AMM interactions. This gives at least a relative measure of the scale of the impact of EIP-3074 on reentrancy, compared to other impacts.

Again in the universe of 648 contracts with a guard combining `tx.origin` and `msg.sender`, we have two analysis versions, one completeness-optimized and one precision-optimized. Their results show:

- 29 contracts in which the guard controls an external call that *may* be reentrant, i.e., with:
    - receiver contract being a function of the caller (including a value derived from a mapping by looking up the caller)
    - without a limit in the gas passed (i.e., no 2300 gas limit, as in Solidity's `transfer`).
- 3 contracts (a subset of the above 29) that in addition have a likely-reentrant call:
    - the call is performed after checking a storage address
    - the same storage address is written-to after the external call.

This analysis is complex and very likely incomplete: the static analysis may not be able to see the semantic connection between the `msg.sender+tx.origin` guard and the external call. However, it is fair to expect that the incompleteness will be analogous to that of the previous case, that of swap calls. (Note that in this case, of reentrancy, we don't have a telltale sign as to which call may be reentrant, whereas Uniswap/Balancer swaps are readily identifiable, since they are standard API calls. Therefore, for reentrancy, we have to speculate as to how many calls are not found by referring to how many were not found in the case of swaps.)

Even with generous assumptions about incompleteness, the numbers suggest that it is highly unlikely that programmers employ the "`tx.origin == msg.sender`" guard as protection against reentrancy. Even considered as accidental protection, the numeric impact seems low--e.g., 3 of 648 contracts (0.46% of contracts with a `tx.origin+msg.sender` guard, or 0.009% of all contracts). Even a doubling of this number (to account for expected incompleteness) suggests a very low impact.

Of course, the real number may be 29 (i.e., the result of the completeness-oriented analysis) and not 3, and, further, this could be 1.5-2x higher if one accounts for incompleteness. However, manual sampling shows this to be unlikely and further confirms that the guard pattern combining `tx.origin` and `msg.sender` is very rarely (if at all) used as reentrancy protection in practice. Specifically, we sampled at random:

- md5 hash 84B1EDCCA6EC726C6E1D99C6F9B23925, deployed twice, at account addresses 0x0FAE0AF7BA4C3AB3527382931B110B0D0C901175 and 0xBA85C3AF2DEA9A5FB1541AC68B92711E19764537. There is no source code, therefore ascertaining the property is hard. However, it seems very unlikely that the guard protects against reentrancy: the two external calls under the guard are to a `transferFrom` function (which should be safe, unless the token is tainted) and to a `_solver` contract, initialized at construction (i.e., trusted).
- md5 hash 934758B4E5877AF9554D717B5B522074, deployed at account address 0x08C82F7513C7952A95029FE3B1587B1FA52DACED. The contract has source. It seems unlikely that the guard offers reentrancy protection, on purpose or inadvertently. There is no other reentrancy protection mechanism in the contract. The calls under the guard are withdrawals, transfers, and swaps. (These should be safe, unless the token is tainted.) The purpose of the guard is clear from a source comment:
  ```
  // Try to make flash-loan exploit harder to do by only allowing externally-owned addresses.
  ```
- md5 hash F1D155B8A6FEDD4FC7E30895A7520E86, deployed at account address 0xC3E2FAFF079BF3864060D61BB892FE21D0D27A93. The guard is certainly not used for reentrancy protection: both functions protected by the guard also have a `nonReentrant` modifier.

Finally, we inspected the 3 warnings of the precision-oriented analysis, for a likely-reentrant call. (This does not mean that the call is reentrant, but that it will be if it is not guarded.)

- md5 hash 1A2D206B826F47A5E2B34BE52842547C deployed twice, at addresses 0x44BD4608AC3BBF8A8677F8B1EA00BD59595F5F9E and 0xECB456EA5365865EBAB8A2661B0C503410E9B347. The guard is not used for reentrancy protection: all its uses also have a separate `nonReentrant` annotation (this is a Vyper contract). In fact, a comment explains the use of the guard:
  ```
  @dev Only callable by an EOA to prevent flashloan exploits
  ```
- md5 hash 39C87BC9D4E18CDD25CB3EB399F7AE6B deployed at address 0x0BD1D668D8E83D14252F2E01D5873DF77A6511F0. This is a Mushrooms Finance contract. There is no threat of reentrancy: the guard is part of a "Keepers" modifier and ensures that a keeper is an EOA. A keeper is trusted anyway and the call that would have been subject to reentrancy is to a trusted "strategy" contract.
- md5 hash 876A2FA9E21AFF9B23C129AA6751BC03 deployed once at address 0xB883F041C0FE3992197FF051644A507C6896C718. The guard is used to allow only EOAs and whitelisted contracts to call `deposit()` and `depositFor()`. The underlying `_deposit()` function employs a reentrancy guard so the `msg.sender == tx.origin` guard is not used as such.

To summarize, by analysis and inspection of contracts that have had any transaction in the past month (200K blocks) we have found zero evidence of use of a `tx.origin+msg.sender` guard that protects against reentrancy, either by design or inadvertently. Given the analysis incompleteness and the limited extent of sampling, it is certainly conceivable that such cases exist (and later manual inspection tasks will uncover some likely instances), but it is virtually certain that they are very rare.

Task 6: Inspect Cases Missed From Automatic Classification

Sample the cases that the above patterns miss. Why do contracts check in practice whether msg.sender is an EOA?

Task 4 showed (at least) 16% of the cases of `tx.origin+msg.sender` guards to be used as protection against price manipulation. Task 5 showed that reentrancy protection does not account for many of the rest of the cases. We next examine, via sampling and manual inspection, what is the likely reason for using the guard in the rest of the contracts. This could be either incompleteness of the earlier automated analyses, or reasons entirely different from price manipulation and reentrancy protection.

Manual inspection can be error-prone and inconclusive. Therefore, the results below should be considered only as an indication, not as hard proof. We include abbreviated auditor's notes and rationale for the classification of each contract, as well as links for easy inspection. (The hex numbers shown are MD5 hashes of bytecode, not contract accounts.)

We randomly sampled 18 contracts and classify them as follows:

Calls other methods of AMMs or interacts with them through other contracts (6 contracts):

- 0E4D7806307C9F2D0BEFED6C184FF0E4(no source): The guard Is used to protect a call to the `rebase` function of the GRPL token. This does not perform a swap but at a later point the `sync` function is called for two uniswap pairs. The guard is very likely intended as protection from price manipulation. Sample tx here.
- 42BB575E700414767386C43AC71DA207: Function `sendToUniswap` has the guard. It is to be called once after a presale is finished and send the gathered ETH to Uniswap using `router.addLiquidityETH()`. It also employs a reentrancy guard so the guard is not used for reentrancy, but likely for price manipulation protection.
- DDA84217A07A06D6524649C421BEC72A (has source on etherscan): BaseTokenOrchestrator of the Base Protocol. `Rebase` function is only callable by EOAs. Is used to call `rebase` on a policy, is the only contract allowed to do so. In the comments there is mention of flash loan protection. Sample tx here. At a later point Uniswap is used.
- 9A316EAF791D54F2C55375BBBD515522: DeFi strategy that has an `onlyBenevolent` guard allowing only EOAs or two approved contracts to call the `harvest` function. Performs swaps. Our decompilation does not include the swaps, hence we do not detect this. Sample tx here. Comments indicate the programmers understand the risks from miner-assisted attacks when deploying it: "*Anyone can harvest it at any given time: I understand the possibility of being frontrun. But ETH is a dark forest, and I wanna see how this plays out. i.e. will be be heavily frontrunned? if so, a new strategy will be deployed.*"
- B6614421BE25FADFA3B330816168BD16: Similar to a previous contract, cannot identify the protocol it is part of (DEBASE?). `rebase()` function on the Orchestrator is callable only by an EOA to prevent flash loan attacks. It then calls `rebase()` on the policy (it is the only one able to do that). Code is complex. Example tx here.
- AB0BAC342397C208403E8841A71AC041: The guard is used optionally inside the `_isKeeper()` function if the `onlyEOA` storage field is set to true. Due to it being optional, in addition to a whitelist, the impact will be minimal. It interacts with Curve. Sample tx here.

No impact (2 contracts):

- 70C590BD31C3B2E22CBE73B619B17A15: Has the inverted guard to only allow smart contracts to send it ETH. Should not have any implications from the EIP.
- 718D4832082247E2C9629598FF476EF2: Has the inverted guard to only allow smart contracts to send it ETH. Should not have any implications from the EIP.

Oracles (4 contracts):

- CEDB64232B850B104B956FBA97CE5FA8: `AccessControlledOffchainAggregator`: Is used to limit the access of view functions to EOAs, as well as approved smart contract callers. Basically, the contract wants to not allow calls from contracts unless they are in a paid "subscribers" list.
- 4DF6E8729641C87316763C6FBA95EC83:
  A variant of `AccessControlledOffchainAggregator`.
- 56A484A6BBB8A2FA605990E45650BB65: Another oracle that allows being called by external accounts or certain whitelisted users. Has many deployments.
- 01F725A98687367D0E153F998865C200: Another oracle.

GSN (1 contract):

- 46B25D3DA3F209F861EE27A4D81AA845: RelayHub contract with 2m usd in ETH. It is used by an Ethereum Gas Station Network (GSN) deployment to relay calls of EOAs. Main GSN documentation does not list it as the deployed relayHub but the one they have is unused. Example tx here. Definitely affected. Reentrancy possibility.

Undetermined (5 contracts):

- 07177F878A839DACF84E15EF645DB40D: (Special Right of Carbon (SRCT) token) The guard is used in public function 0xa044c987 to withdraw ETH (using a 2300 gas transfer). The withdrawable ETH amounts for each `msg.sender` are only assigned by the owner using the `manage` public function. Due to the above the guard does not seem to prevent reentrancy, due to only a low-gas transfer being used. Has never been used after its deployment so our best guess would be that guarding is done to ensure the transfer will succeed.

- 2F1351269073AB9F41B3DE5EAE9AADEB: Using `tx.origin == msg.sender` to treat contracts and EOAs differently but not in a consistent way. There is also a smart contract `greyList` used so the check is bypassable at points.
- EEECF03826A0F7E30D9CA970745698ED: Spore finance pool (SporePool.sol, uses Defensible.sol for the described protection). Only allows EOAs and accepted contracts to stake and vote.
- 1353EAE735F645AFD7D931F202CB8AC1: nordUSDT of nord finance according to Etherscan. No source available. Had trouble getting a good decompilation. Panoramix also fails to decompile the most difficult functions. The guard is used for the `deposit()` and `depositFor()` functions. Looked for transactions to understand what these functions do. Only got some for deposit.
- 77B213B4A156A494828067AB4E0C32EB (no source): Something to do with Hoopoe finance. Protects public functions (click for sample txs) `0x822b2f9d`, `0x88f1bf5e`, `InvestERC20`, `InvestETH`. Mainly performs transfers of Hoopoe finance tokens HOOP, ETHXAU, ETHXAG.

In summary, manual inspection of a sample of 18 contracts with the `tx.origin+msg.sender` guard confirms that the dominant usage scenario is protection from flash loans/price manipulation. This scenario accounts for 6 of the 13 "determined" cases. 2 more cases are determined to not be impacted by EIP-3074.

Of the remaining 5 cases, a common pattern (4 cases) is that of oracle contracts that allow free access to EOAs. These contracts are semantically affected by EIP-3074. However, since the contracts already have access lists for contract callers, they can adapt relatively easily.

The most interesting case is the GSN contract. It is semantically affected and has a likely reentrancy vulnerability under EIP-3074. However, GSN is a transaction relayer. It will certainly need to change to adapt to stay relevant (and likely to evolve/improve) under EIP-3074.

Task 7: Automated Query of Potential Impact

Combine all the above queries with severity metrics. How many of the contracts in each category have funds or approvals? How many seem to be trusted by other contracts with funds or approvals (based on behavior in past transactions)?

The next task aims to see if the impact can be quantified via automated severity queries, based on account balances and token approvals granted to the likely-impacted contracts. The queries are again over the set of 648 unique contracts (1.85% of those that transacted in the last 200K blocks).

The queries will only give an upper bound of the impact. Unfortunately, this upper bound does not turn out to be tight at all. There are large sums in the balances or allowances of likely-impacted contracts, however they are unlikely to be truly impacted by the EIP-3074 change. For instance, for yield farming protocols, it is very likely that the daily/periodic yield is swapped and this "harvest" is EOA-guarded, yet the amount of the swap is minuscule compared to the contract's overall balances or approvals.

The 648 contracts yield over 2400 rows of ERC-20 token balances, based on past transfer events. We inspected around 100 balances that correspond to large transfer events. We list below some accounts that contain the `tx.origin+msg.sender` guard and hold large balances. Obvious false positives (i.e., instances of "`tx.origin != msg.sender`") have been removed.
(Since balances pertain to accounts and not to contracts, we list addresses and not MD5 hashes below, and link to Etherscan for inspection.)

- 0x86690BBE7A9683A8BAD4812C2E816FD17BC9715C
  1.1M (DeFi Yield Protocol)
  They use it to guard functions that perform swaps.
- 0x78E2DA2EDA6DF49BAE46E3B51528BAF5C106E654
  1.4M (DeFi Yield Protocol)
- 0x1DCAF36C8C4222139899690945F4382F298F8735
  8M (CompoundWETHFoldStrategy for Harvest)
  Sample tx here. Shows that liquidity is removed from Uniswap at a later point.
- 0x3FB6B07D77DACE1BA6B5F6AB1D8668643D15A2CC
  2.1M (Bella Wrapped BTC coin)
  Used explicitly in earn() but is also part of the onlyWhitelisted modifier which is used in a bunch of places. Sample earn tx here.
- 0x924BECC8F4059987E4BC4B741B7C354FF52C25E4
  0.8M (DeFi Yield Protocol)
- 0xFC89086C0B1F8ACBD342F418D3EA1C9E425E5CBB
  650K (Mushrooms Fi)
  Used to protect the harvest function, which performs a swap.
- 0x4A76FC15D3FBF3855127EC5DA8AAF02DE7CA06B3
  313K (DeFi Yield Protocol)

- 0x350F3FE979BFAD4766298713C83B387C2D2D7A7A
  2.0M (DeFi Yield Protocol)
- 0x8B0E324EEDE360CAB670A6AD12940736D74F701E
  646K (DeFi Yield Protocol)
- 0x5E3E0548935A83AD29FB2A9153D331DC6D49020F
  209K (unknown). Uses uniswap as an oracle. Flash-loan protection.
- 0x2DE9441C3E22725474146450FC3467A2C778040F
  200K (Wasabix finance)
- 0x2B5D7A865A3888836D15D69DCCBAD682663DCDBB
  307K (DeFi Yield Protocol)
- 0x0B0A544AE6131801522E3AC1FBAC6D311094C94C
  477K (DeFi Yield Protocol)
- 0xEF71DE5CB40F7985FEB92AA49D8E3E84063AF3BB
  497K (DeFi Yield Protocol)
- 0xA52250F98293C17C894D58CF4F78C925DC8955D0
  469K (DeFi Yield Protocol)

For allowances, the upper bound is even more loose. We found 92K rows of approvals for likely-impacted contracts and some contracts have very large approvals--e.g., the following Harvest Finance contract: 0xF8CE90C2710713552FB564869694B2505BFC0846. The approved sums are overwhelmingly not impacted by the EIP, however.

In summary, we consider the queries of Task 7 to have been interesting but rather uninformative. The likely-impacted contracts hold large sums, but these sums are at most very partially exposed to the impact of EIP-3074. The next task sheds more light onto this aspect.

Task 8: Manual Inspection of Major Projects, Anecdotes

Manually inspect contracts to determine impact. E.g., are there well-known services that are impacted and how?

For this task, we have cloned the code repositories of a large array of protocols, including several of the most prominent in the ecosystem. We search for `tx.origin+msg.sender` guarding patterns and inspect their use. We also discuss some anecdotes.

All protocols examined (informal naming, as on our disk):

0x-monorepo, **1inchProtocol**, aave, airswap-protocols, **alchemix**, alpha finance, **armor**, **augur**, **badger**, balancer, bancor, **bt-finance**, **bZx**, chainlink, charged particles, charm finance, **compound**, cover, **cream**, cryptomaniacs-mooniswap, curve, defi dollar, defi-saver, **delta-financial**, dydx, element finance, fei protocol, furucombo, gnosis, **gsn**, **harvest-fi**, horizon, hydro, idle, instadapp, **kyber**, liquity, loopring, maker, maple, **mStable-contracts**, **mushroomsFi**, nexusmutual, nuo, opyn-GammaProtocol, origin-dollar, paid, primitive, rari, ren, **set-protocol-v2-contracts**, **Sovryn-smart-contracts**, **sushiswap**, SwerveContracts, synthetix, uniswap, unslashed, vesper, warp finance, **yaxis**, **yearn**.

The repositories that syntactically match the guarding pattern (`tx.origin == msg.sender`, or variations) are shown in **boldface**--*the rest are entirely unaffected*. Below we include comments from the inspection of several of these matches. (We color-code flash-loan protection with pink, other impact with orange, undetermined with yellow, and unaffected with no color. A single protocol may contain contracts affected in different ways.)

A. Alchemix
   Used in the `noContractAllowed` modifier of Alchemist.sol. Probably flash loan protection. Also it's always used with a reentrancy guard. Deployed here. Sample txs of the protected operations here, here, here, here.
B. Alpha finance
   Used in:
   ./alphahomora/contracts/5/MStableGoblin.sol,
   ./alphahomora/contracts/5/UniswapGoblin.sol,
   ./alphahomora/contracts/5/SushiswapPool12Goblin.sol,
   ./alphahomora/contracts/5/Bank.sol,
   ./alphahomora/contracts/5/SushiswapGoblin.sol:
   In all the above cases the guard is used in an `onlyEOA` modifier with its motivation being flash loan protection, as indicated by the comments. Also it's always used with a reentrancy guard.
   Bank deployment with sample "work" transaction shows it adds significant amounts to uniswap liquidity. Having trouble finding the deployments of other contracts.
   Additionally also used in ./homora-v2/contracts/HomoraBank.sol.
   Their docs talk about "*No more EOA only = more composability*" so they seem to be moving away from such patterns.
C. Armor (cloned arCore, arNXM)
   Used in arNXM with a comment "Avoid composability issues for liquidation."

"For our contracts [EIP-3074] won't have too much of an effect." (Robert Forster, Armor co-founder and lead developer)

D. Badger (cloned badger-system):
Used in:
./badger-system/contracts/digg-core/Orchestrator.sol:
Used as flash-loan protection of the `rebase()` function as shown by the test comments. Found it deployed here. Sample tx here. Shows that it interacts with uniswap later calling a `.sync` function. Also part of their documentation talks about when rebase can happen. It involves multiple actions.
./badger-system/contracts/badger-sett/SettAccessControlDefended.sol:
Used in `_defend()` of SettAccessControlDefended, employed in various Sett files. This is done for flash-loan protection. Here they talk about this as well as another mechanism they employ for flash loan protection.

E. bt-finance (cloned bt-finance)
Used in:
./contracts/vault/bVaultV2.1.sol
./contracts/vault/bVaultETHV2.1.sol
Very likely flash loan protection.

F. bZx (cloned contractsV2)
Uses:
contractsV2/contracts/staking/FeeExtractor_BSC.sol,
contractsV2/contracts/staking/StakingV1.sol:
Used in `onlyEOA` modifier protecting `sweepFeesByAsset`, which performs swaps. Doesn't have any reentrancy protection but it looks like the sender cannot intercept control flow. Cannot find it deployed on their list.

contractsV2/contracts/modules/LoanClosings/LoanClosingsWithGasToken.sol,
contractsV2/contracts/modules/LoanClosings/LoanClosings.sol:
Used in both cases to protect a rollover function, the guard says "*restrict to EOAs to prevent griefing attacks, during interest rate recalculation*". Used together with a reentrancy guard.

G. Chainlink (cloned chainlink):
./evm-contracts/src/v0.6/FluxAggregator.sol
./evm-contracts/src/v0.6/SimpleReadAccessController.sol
These correspond to the examples of oracles we saw earlier. They limit access to oracle results to EOAs and accounts(contracts) added to an access list. With the proposed changes of the EIP any contract could bypass this.

H. Compound (cloned compound-protocol):
Used in:

./compound-protocol/contracts/ComptrollerG1.sol
./compound-protocol/contracts/ComptrollerG5.sol
./compound-protocol/contracts/ComptrollerG4.sol
./compound-protocol/contracts/ComptrollerG6.sol:
All these appear to be old development versions and not deployed.

I. GSN (cloned gsn)
Used in packages/contracts/src/RelayHub.sol. It protects `relayCall`, the main entry point of GSN's RelayHub. It keeps the system's invariant that the relayers are EOAs. With a brief look it seems reentrable. The EIP is related with GSN's functionality so it is natural that GSN would evolve, as also discussed in task 6.

J. Harvest Finance (cloned harvest, harvest-strategy, harvest-strategy-bsc):
Used a lot, citing flash-loan protection.

K. Set-protocol (cloned set-protocol-contracts, set-protocol-strategies, set-protocol-v2):
Uses:
set-protocol-v2/contracts/product/UniswapYieldHook.sol:
Used in `invokePreIssueHook`, `invokePreRedeemHook` to limit the amounts contracts can issue or redeem. Flash-loan protection.

set-protocol-v2/contracts/protocol/modules/GeneralIndexModule.sol,
set-protocol-v2/contracts/protocol/modules/SingleIndexModule.sol:
`onlyEOAIfUnrestricted`, `onlyEOA` modifiers, optionally protecting some trade functions. Alternatively has whitelists. Perform swaps through some abstraction for exchanges. Always used together with reentrancy guards.

L. Sovryn (cloned Bridge-SC, oracle-based-amm, Sovryn-smart-contracts):
Used in Sovryn-smart-contracts/contracts/modules/LoanClosingsBase.sol, citing griefing protection, also has reentrancy guard.

M. Sushiswap (cloned Inari, kashi-lending, LimitOrderV2, mirin, sushiswap, sushiswap-settlement, sushiswap-zapper)
Used in:
sushiswap-settlement/contracts/Settlement.sol,
sushiswap/contracts/SushiMaker.sol,
sushiswap/contracts/SushiMakerKashi.sol:
Used to prevent flash-loan attacks, they talk about it not being perfect.
Deployments SushiMaker and SushiMakerKashi. Former is used way more.
sushiswap-zapper/contracts/Sushiswap_ZapOut_General_V1.sol:
Inverted guard is used to avoid accidental eth transfers from EOAs.

N. Yaxis metavault
Used in ./metavault/yAxisMetaVault.sol. At the `checkContract` modifier protecting

`deposit` and `depositAll`. Its use is optional with a boolean `acceptContractDepositor` field indicating if contracts are allowed or not. Their FAQ discusses how this is done to protect against smart contracts: *MetaVault simply disallows smart contract interaction with the vault, which prevents flash loan exploits and other common attack vectors. We can also run simulated exploits in real-time to test our systems.*

O. Yearn (cloned yearn-protocol, yearn-vaults)
Only used in yearn-protocol/contracts/strategies/StrategyYFIGovernance.sol for flash-loan protection, swap performed on the same tx. According to their documentation it is deployed here. Hasn't been used for 90 days. Before that it would be called every 3 days and result in the swap of small sums (max we saw was 6k).

The inspection of prominent protocols confirms the findings of earlier tasks. There is a significant number of protocols that contain at least one contract with a `msg.sender+tx.origin` guard (some-20 protocols out of the ~60 examined), some in multiple ways. (In terms of proportion of affected *contracts*, the number would be much smaller, since these protocols are large. There are several thousand .sol files in our directory of cloned repositories, although this number also includes interfaces and contracts duplicated across projects. Conversely, it is possible that a single impacted contract affects a large part of the entire protocol's functionality.)

Reentrancy protection is extremely rare (possibly only in GSN). Most instances of the guard (10 of the 17 contracts found to be impacted) are intended for flash loan protection. Other affected patterns include *oracles* (i.e., a business decision regarding pricing of services for EOAs vs contracts, as discussed in task 6) and *griefing protection* (ensuring that an ETH transfer cannot fail).

Griefing protection is interesting. Although rare (2 instances), it is a pattern we had not anticipated.

Anecdotally, programmers seem aware of the dangers of guarding with "`msg.sender == tx.origin`" and anticipate this protection to be imperfect. One can often find comments of the form

`// At least we try to make front-running harder to do.`

or

`// Try to make flash-loan exploit harder to do by only allowing externally owned addresses.`

Generally, it is safe to say that the guarding pattern in question is often added to the code out of an abundance of caution, i.e., "just because it's easy extra protection". Even in code that cites, e.g., griefing or flash-loan protection, we have often been unable to find a real attack that could have impact under EIP-3074.

Opinion

Based on the above querying, analysis, inspection, and informal communication, it is clear that the impact of EIP-3074 is nominally significant. The greatest impact is on the check "`msg.sender == tx.origin`", which is common as protection against price manipulation, in interactions with an AMM. The check currently ensures that the caller is an EOA, but this will no longer be the case if EIP-3074 (in its current, "strong" form) is adopted. (There are also other isolated impacts, e.g., to GSN or oracle contracts, but they are relatively rare.)

The severity of this change, however, depends on the security that the guarding pattern currently affords. It is our opinion that the "`msg.sender == tx.origin`" pattern will not afford security against price manipulation for long, independently of EIP-3074.

Specifically, miners (or attackers colluding with miners) already sandwich transactions, effectively turning multiple transactions into a single, indivisible one, in order to reap profits. In this sense, it is irrelevant whether the `msg.sender` of a call is an EOA: what is important is whether the on-chain price of an asset could have been manipulated atomically before the call (without any opportunity of market correction).

The current state of affairs as of this writing (May 19, 2021) is that miners themselves routinely conduct sandwiching attacks against victim transactions, as long as the transaction is "simple" (e.g., one side involves ETH only). We have witnessed attacks with sums well over 1000x larger than the eventual profit. It is only a matter of time before miners start performing complex sandwich attacks with more than ETH (i.e., with arbitrary ERC-20 tokens). It is possible that they have not done so on a major scale until now only because of the sophistication needed to counter the threat of loss (e.g., poisoned tokens). We believe it is also a matter of time before miners start performing such attacks proactively, i.e., without seeing a victim transaction, just by identifying code that anyone can call and that can provoke a swap.

Even more sophisticated attacks are currently being routinely performed by third-party attackers who collude with miners (using flashbot/MEV miner bribes). Such sandwich attacks involve both ETH and tokens. We are not aware if there are instances of proactive attacks (i.e., crafting EOA transactions and sandwiching them, rather than sandwiching an existing one) but it seems just a matter of time for this to happen. The only obstacle seems to be the need to identify victims and craft attack transactions. Third-party attackers are likely to have even fewer qualms than miners to employ a proactive attack, yet the sandwich attack will enjoy atomicity by leveraging miner collaboration.

It seems that in the future these brazen attacks will become even more commonplace. Effectively, the difference between a “guaranteed atomic” flash-loan attack and a miner-facilitated “in-effect-atomic” plain-loan attack is already greatly diminished.

Finally, one can ask if an attack might a) be profitable only after EIP-3074, or if b) it might be easier or more devastating after EIP-3074. The former is likely not the case, the latter is likely the case.

For question (a), we can consider swaps as the common example of a victim transaction. (Other attacks, such as using distorted pricing for issuing more “shares” are similar.) Whether a pool-tilting attack is profitable or not is mostly independent of the sum available to attack (modulo attacks of such low value as to be rendered non-profitable due to gas costs). The rule of thumb is that a pool-tilting attack is profitable if the amount being swapped by the victim is greater than the price of a one-directional swap (e.g., 0.3% for Uniswap v2 pools). If this condition is satisfied, any tilting of the pool is profitable for the attacker.

For question (b), flash-loan attacks are easier and can potentially be more devastating than attacks with actually-held (or borrowed-from-miner) funds. Flash-loans require no up-front holdings and can tilt the pool by many tens of millions of dollar-equivalent value, whereas typical miner-assisted attacks are for at most a few million and the miner needs to be holding this amount. However, this does not change the fact that the attack is profitable and that miners have already been performing large attacks for small profit.

In summary, code that employs the “`msg.sender == tx.origin`” pattern is already not safe. An attack may become easier with EIP-3074, but the rate of evolution of attacks (especially in collusion with miners) does not allow labeling this EOA check as anything but an anti-pattern.

Conclusion

We studied the impact of EIP-3074 on existing contracts. The impact is not negligible, with several thousand of deployed contracts expected to be affected (at a sampled rate of 1.85% for contracts with recent transactions). These contracts handle significant sums and may expose them partially to vulnerabilities. However a) the community of developers seems well-aware of the potential risks and ready to adapt with sufficient warning; b) affected code is already subject to attacks that are likely to become commonplace in the near future (with the latter being a subjective assessment). As a result, we rate the impact of EIP-3074 as “moderate but manageable”, yet acknowledge that the results are subject to interpretation.

Disclaimer

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The study is performed on a best-effort basis: Dedaub offers no promise or guarantee as to the accuracy of the findings, either in terms of completeness or in terms of precision. There may be contracts impacted that are missing from the report, as well as contracts included that, due to semantic complexities, are unaffected.

About Dedaub

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.