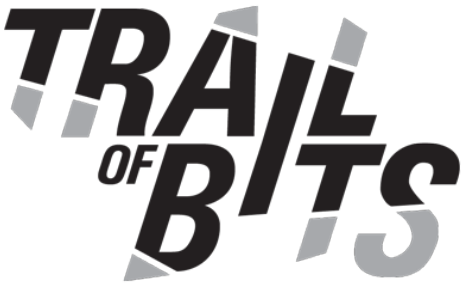

EIP-1283 Incident Report

Background and Lessons Learned

Prepared For:
Hudson Jameson | *Ethereum Foundation*
hudson@ethereum.org

Martin Swende | *Ethereum Foundation*
martin.swende@ethereum.org

Prepared By:
Dan Guido | *Trail of Bits*
dan@trailofbits.com

Josselin Feist | *Trail of Bits*
josselin@trailofbits.com

David Pokora | *Trail of Bits*
david.pokora@trailofbits.com

Evan Sultanik | *Trail of Bits*
evan.sultanik@trailofbits.com

Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Petar Tsankov | *ChainSecurity*
petar@chainsecurity.com

Matthias Egli | *ChainSecurity*
matthias@chainsecurity.com

Hubert Ritzdorf | *ChainSecurity*
hubert@chainsecurity.com

Tomasz Kolinko | *Eveem*
kolinko@gmail.com

Date of Report: January 17th, 2019
Document Version: 1.2

Changelog

1.0 (January 16, 2018): Initial release
1.1 (January 16, 2018): Added further detail to recommendations
1.2 (January 17, 2018): Added Appendix B and Appendix C

EIP-1283 Gas Calculation Security Incident

Background

EIP-1283 was initially proposed on August 1, 2018. It was accepted on November 28, 2018. On January 14th, 2018, two days before the Constantinople hard fork, a security issue in EIP-1283 was discovered and detailed by ChainSecurity. The fork's modification to gas computation would have led previously safe contracts to become vulnerable to a reentrancy attack. For approximately 24 hours spanning January 15th to 16th, 2018, Trail of Bits, ChainSecurity, and Eveem proceeded to assess the number of vulnerable contracts on Ethereum mainnet, considered potential mitigations, and reflected on how the situation could have been avoided.

The current EIP process depends on one of a few security experts dedicating enough of her time to fully review each modification. Security review may not always occur for EIPs that need it.

The security of Ethereum smart contracts relies on the immutability of the blockchain. A contract's properties are typically proven safe for only a single specification of the EVM. Any change to the EVM specification can lead to a violation of an existing contract's general semantics, and thereby its security properties. EIP-1283 constituted such a change.

Vulnerability Description

A common reentrancy mitigation relies on the use of the Solidity `send` and `transfer` functions. These functions limit the gas given to a call to 2300. It is assumed that it is not possible to change a state variable with this gas limitation. EIP-1283 introduces a new gas computation which allows for cheaper state variable modifications. This makes it possible to change a state variable with less than 2300 gas, breaking the reentrancy mitigation used by already deployed contracts.

See ChainSecurity's announcement for further explanation, including a proof-of-concept exploit.

Exploit Scenario

Alice develops a smart contract and ensures the safety of the code through formal methods. The contract is proved reentrancy-safe. Alice deploys the contract and transfers 100 ethers to it. The Constantinople hard fork occurs. Gas computation rules change. Alice's contract becomes vulnerable to a reentrancy attack. Bob takes advantage of the reentrancy and steals the 100 ethers.

Recommendations

Short-term

Do not implement EIP-1283. Low gas cost for storage changes can break the implicit assumption that `CALL` instructions with the 2300 gas stipend are safe from reentrancies.

If EIP-1283 is implemented as originally specified, ensure that dirty storage is tracked per call, not per transaction. This will ensure that any call causing reentrancy will not be given a gas discount. Consider using a new opcode to prevent changing the behavior of existing contracts.

Consider refunding the difference of the intended cost of writes, detailed in Appendix C.

Long-term

Explicitly define which parts of Ethereum are considered immutable and which ones may change in future network upgrades. For example:

Component	Mutability	Discussion
Instruction Semantics	Immutable?	Instruction semantics must be immutable.
Available instructions	Mutable?	Instructions can be added but must not be removed.
Gas cost of instructions	Immutable?	Consider denial-of-service attacks, and whether gas costs could be restricted to move only in a single direction (*e.g.*, up). Favor new opcodes rather than changing the gas cost of existing ones. Changing the gas cost of instructions would not be a problem if developers did not assume they are invariant from one version of EVM to another.
Gas limit per transaction/block	Mutable?	Can this decrease over time?

Define a set of criteria to determine “high-risk” EIPs. For any high-risk EIPs, allocate specialized security resources to fully review their ramifications prior to approval. Consider

as high-risk any EIP that changes existing contract behavior. In particular, flag for security review EIPs that reference the following:

- Modifications to gas computation
- Contract upgradability
- New instructions that lower gas usage for existing functionality[1]
- New instructions that introduce a new calling convention
- Change to value transactions, especially Ether

Consider creating a reentrancy-safe CALL instruction that can be used when reentrancy is undesired. This instruction would allow a contract to send ethers to another contract, execute some instructions (such as LOG), but prevent a reentrancy attack. A reentrancy-mitigation at the EVM level would prevent incorrect assumptions from high-level languages, such as Solidity and Vyper. Alternatively, consider adding a recursion-limiting opcode that prevents a contract from recurring on the call stack thereafter.[2]

Review and update documentation to underscore sensitive areas of Ethereum that could retroactively change how earlier deployed smart contracts behave. For instance, developers are often told to assume that transfer/send operations are reentrancy-safe due to gas costs, when gas cost is a property of Ethereum that could be subject to change.

Consider specifying a “constitution” that defines what can be changed between different versions of the EVM. This would allow developers to begin using patterns that are future-proofed.

[1] This may enable contracts to conduct attacks that were previously impossible, *e.g.,* `"CALL_CHEAP"`.
[2] Recursion in this context is meant on a contract-level, not a function-level.

Appendix A. List of vulnerable contracts

This non-exhaustive list of contracts would have been vulnerable[3] to the reentrancy attack:

testingToken

- 0x41dfc15CF7143B859a681dc50dCB3767f44B6E0b
- 0x9c794584B2f482653937B529647924606446E7F4
- 0x911D71eEd45dBc20059004f8476Fe149105bF1Dc
- 0x693399AAe96A88B966B05394774cFb7b880355B4

Artwork

- 0x98eA61752e448b5b87e1ed9b64fe024B40c6127d
- 0x4f1DcdAbEEA91ED4b6341e7396127077161F69eD
- 0xa3cE9716F5914e6Bb5e6F80E5DD692d640F8608c
- 0xC82Fe8071B352Ee022FaB5064Ff5c0148e3ac3aa
- 0x95583A705587EDed8ecBaF1E8DE854e778f366C4
- 0x1FCC17b8e72b65fD6224ababaA72128D2153C1FA
- 0xc14971b19a39327C032CcFfBD1b714C0F886dc76
- 0x626e6a26423ce9dd358e1e5bd84bce01de07bc73
- 0x22164E957ac4C0cB0f19C49B05e627675436DFE1

We scanned the blockchain with several tools including Eveem, ChainSecurity's Symbolic Verifier, and Trail of Bits' Slither analyzer. The vast majority of the contracts did not present an immediate risk. In several contracts where a reentrancy is possible with EIP-1283, the reentrancy did not result in a violation of a contract-specific property. Nonetheless, EIP-1283 does change the semantics of such contracts.

Several code patterns (such as upgradability and heavy use of reference storage) are challenging for the automated analyses, increasing the likelihood of missing bugs. Our approach could only find the presence of vulnerable contracts and is not suited to find all the instances of the bug.

We did not find any vulnerable high-profile contracts during the limited time of research. However, we did discover real instances of the bug on mainnet, and we found contracts that could have been vulnerable if the generated bytecode had been optimized. If deployed, EIP-1283 would likely have been the enabler of a reentrancy hack.

[3] *E.g.*, EIP-1283 allows the contract to reach a state that was previously not possible.

Appendix B. Review of proposed remediations

On January 15th, a thread was started on the Fellowship of Ethereum Magicians forum to collect proposed remediations. This resulted in six proposals which we review below.

All of the six proposals are incomplete or introduce undesirable changes in existing contracts. We recommend selecting one of the following paths forward:

- Abandon EIP-1283 and avoid implementing it at all.
- Implement our own proposal #7 to address the security risks of EIP-1283.

The following describes our current analysis of the propositions. Our main concerns toward the existing propositions are:

- Proposition #1, #3 and #5 change the semantics of existing contracts.
- Proposition #2 is a good long term recommendation, but does not fix the issue.
- Proposition #4 breaks existing contracts.
- Proposition #6 has potential implications that are not, for now, thoroughly analyzed.

Proposal 1: Add a condition to SSTORE that reverts if less than 2300 gas remains

This proposal is meant to capture reentrancies due to Solidity `send` and `transfer` functions, which provide only 2300 gas.

This proposal changes the semantics of existing contracts. Moreover, it does not prevent reentrancy for calls with gas between 2300 and the minimal SSTORE gas price of 5000. A contract providing 4999 gas would be safe from reentrancy based on state variables changed *prior* to EIP-1283, but would become vulnerable with this proposition.

We do not recommend this proposal.

Proposal 2: Add a new call context that permits LOG opcodes but not changes to state

This proposal creates a new opcode which is meant to be reentrancy-safe.

The proposal follows one of our long-term recommendations. While it *does not* mitigate EIP-1283, it *will* provide a safer way for high-level languages to send ethers to an address, or allow developers to specify that a call should not re-enter.

We recommend following this proposal as a future work.

Proposal 3: Raise the cost of SSTORE to dirty slots to >=2300 gas

The proposal raises the gas cost of a dirty slot to be greater than or equal to 2300.

In addition to having the same weakness as Proposal #1, it removes part of the usefulness of EIP-1283 by having a more expensive SSTORE, which makes the EIP less interesting to implement.

We do not recommend this proposal.

Proposal 4: Reduce the gas stipend sent

This solution will either break existing contracts (if the gas stipend sent is changed for all the contracts) or fail to protect them.

We do not recommend this proposal.

Proposal 5: Increase the cost of writes to dirty slots back to 5000 gas, and add 4800 gas to the refund counter

This proposal aims to provide the same execution gas price for changes to dirty storage while providing a discount through the refund mechanism. This mechanism is meant to prevent the reentrancy. Unfortunately, the proposal may have consequences on already deployed contracts.

For example, the following cases will change the semantics of existing contracts:

- A no-op storage to a dirty storage will cost 200 gas instead of 5,000
- A dirty storage with an original cost of 20,000 will cost 5,000

These changes might lead to unexpected behaviors in already deployed contracts.

We do not recommend this proposal.

Proposal 6: Add contract metadata specifying per-contract EVM version, and only apply SSTORE changes to contracts deployed with the new version.

This proposal introduces versioning to the EVM. Contracts will follow the specification of the EVM according to their time of deployment.

This proposal introduces many corner cases that must be thoroughly reviewed. If a contract deployed *before* a fork creates a new contract *after* the fork, to which EVM specification should the new contract be bound? How will inter-contract interactions like delegatecall work between contracts deployed with different EVM specifications?

Versioning might be problematic for smart contracts that require several months of development and could potentially need to be verified against multiple versions.

Versioning will increase the complexity of Ethereum clients in such a way that might lead to subtle edge cases and implementation differences that could lead to consensus issues.

Finally, versioning will also increase the complexity of the formal analysis tools, as they will need to precisely model contracts' interactions with respect to different semantics.

We recommend a thorough analysis of the impact of this proposal.

Appendix C. Proposal #7: Refund difference of intended cost of writes

Integrity of execution should be maintained to prevent re-entrancy, and to avoid introducing new code paths that would have been unreachable before the gas refund changes. We should not retroactively affect previously deployed contract behavior, *if possible*. Of course, exceptions such as the Tangerine Whistle/Anti-DOS fork exist, as there was no other remediation.

Charging the same pre-EIP-1283 gas cost but providing the intended EIP-1283 discount through refunds may preserve the original execution semantics and code reachability. This would work because refunds are currently awarded after the transaction has completed. For example, if a storage operation cost 20,000 before the proposal, but we wish to charge the user only 5,000, we can deduct 20,000 gas and refund the difference (15,000).

Currently, a refund cannot exceed more than half of the gas used per transaction. We may want these proposed refunds to bypass this upper bound in order to fully take advantage of intended deductions.

This proposal *both* maintains the integrity of execution *and* provides the intended discounts. However, it requires users to send the same amount of gas upfront for the transaction to succeed, and refunds to be awarded.

If the requirement of providing the original gas cost upfront is acceptable, we recommend using this solution to achieve gas savings as in EIP-1283.