Co-op Work Report: Gas-related security issues in Ethereum

Sung-Shine Lee

Wednesday 16th October, 2019

1 Introduction

Blockchain, the technology behind Bitcoin and other cryptocurrencies, has gathered attention since Satoshi had proposed the Bitcoin on 2009. While the Bitcoin protocol was designed to be used as a currency and its scripting ability was intentionally limited to prevent trivial malicious behaviour (such as having an infinite loop in the script), people have soon shown interest in extending the ability of Bitcoin with techniques such as colored coin to make these scripts, or smart contracts, have more ability. Vitalik Buterin proposed the first Turing complete blockchain system Ethereum, allowing all kinds of smart contracts to be run on the decentralized system, at the same time defending against infinite loops by introducing the concept of gas as the transaction fee. Gas is what the transaction sender would send along his/her transaction, and every operation that is executed in the smart contract will consume some gas. By introducing gas, the underlying machine can be Turing complete, but at the same time, smart contract executions are limited by the resource of the transaction initiator.

As all programs in other fields, smart contracts are typically written by humans and are error prone. What makes it worse than the typical software errors is that smart contracts are usually immutable, therefore one can not patch the program post deployment. Furthermore, smart contracts are often used to control funds on the blockchain: Indeed, the history of smart contracts are full of hacks that involves large amount of capital, e.g. the loss of the DAO hack in 2016 is around 60 million USD. It became clear that smart contract security is a pressing issue that needs to be addressed. The field has since grew fast, as resources and efforts are put into enhancing the security of smart contract. Smart contract development is one of the fields where developers or auditors are using formal methods, static analysis, and other tools during development to assist them in finding vulnerabilities before deployment.

In the recent development of Ethereum, notably the Istanbul update, includes 4 Ethereum Improvement Proposals (EIPs) related to changing the gas price. Historically, gas price changes has broke smart contracts and therefore it would be essential to understand the changes and what effects they would have on deployed contracts.

2 Methodology

Working as a security auditor in the blockchain field, this report approaches the gas related security issues in the Ethereum network. I would first review gas related security issues that has been uncovered in the past to gain an overall understanding. Second, I would discuss the gas related EIPs that are accepted into the Istanbul hard fork. Third, security implications of these updates will be presented to gain a better understanding. Finally, I conclude with some security recommendations for developers and future works that needs to be done regarding the gas price update.

3 Introduction to gas in Ethereum

Ethereum incentivizes miners to verify and execute transactions by compensating them with mining rewards and transaction fee. While mining rewards are predetermined in the protocol, transaction fees are not. As miners executes smart contract for transaction senders, it is important that they are incentivized to execute the contracts and malicious transaction senders cannot attack the network via having the miners running infinite loops. Ethereum introduced the concept of gas when calculating the transaction fee, and designed it in the way such that every operation executed in the smart contract would consume some gas.

The transaction sender has to send the maximum gas to be consumed and gas price along with the transaction. The miner would keep track of the gas consumed and convert to amount of ether with the gas price specified by the sender in the transaction.

In Ethereum, there is a limit of gas in which a block can consume. Therefore a miner can only include limited amount of transactions in a block, as a consequence, miners would in general select transactions with high gas values to maximize their profit.

4 Analyzing gas related security issues in the past

4.1 Network level DoS : Hitting the block gas limit

As it is natural for the miners to select transactions with high gas values to maximize their profit and there is a limit on the total amount of gas to be consumed in a block, it is possible for a transaction sender to exploit miners' behaviour and temporarily DoS the network with some monetary costs. A malicious transaction sender can craft transactions that do meaningless loops to consume gas, and provide the transaction with a high gas price to ensure that it would be included by the miners. These transactions can be crafted to reach the maximum gas limit in a block, so that no other transaction can be included by the miners. The whole network thus only works on the transactions provided by the attacker, and appears to be stalled by everyone else.

Although the attacker would pay a price to launch such attack, it is possible that one could profit in such cases if enough incentives are present. For example, a Ponzi scheme smart contract that accumulates users' bid, FOMO3D, was deployed on the Ethereum. FOMO3D is designed to reward the last bidder with all the capitals accumulated in the smart contract if there is no follow-up bidders in a couple of blocks. The attack above was successfully launched to prevent other bidders within the number of blocks. The attacker were able to stall the whole network and profit from winning the bid.

4.2 Trick message sender into burning unnecessary gas

Sending ethers to a smart contract would invoke its fallback function, therefore it is possible for the attacker to craft the fallback function so that it consumes a lot of gas. A typical application would often times automatically set the gas limit of a transaction by first simulating the transaction and estimate the amount of gas that would be consumed. If no measures were taken, it is possible for the attacker to trick the application to send some ether to a certain address, and burn an excessive amount of gas. While the attack can only cause damage to the message sender by itself, it has been discovered that an attacker can combine the technique with the mechanism of getting refunds when deleting storage on Ethereum to make a profit.

Here's an example fallback function that burn sender's gas:

```
function() external payable {
    for (uint i = 0; i < 1000; i++) {
        DoSomeMeaninglessOperations += 1;
    }
}
```

4.3 Contract Level DoS: Growing arrays, growing gas consumption

Benevolent programming patterns in other languages may sometimes be harmful in Ethereum. For example, developers are used to use loops to process arrays that might grow overtime. Note that every execution in Ethereum costs gas, therefore the gas consumption of such loops would also increase overtime. Ethereum developers often overlook this issue and allow the array to grow indefinitely without contemplating on how might the numbers grow. In some cases, the gas consumption might overgrow the block gas limit, leading to a DoS situation of the contract.

```
function growArraySize(targetData){
  unknownSizeArray.append(targetData);
}

function updateData( target, updateTo) {
    for (uint i = 0; i < unknownSizeArray.length; i++) {
        if ( unknownSizeArray[i] == target ){
           unknownSizeArray[i] = updateTo;
        }
    }
}
```

4.4 Network level DoS: Mismatch of opcode gas pricing and real execution cost

Since gas was designed to compensate miners for performing computation, if the price mismatches the actual execution cost, the network protocol is unfair since either the miners will gain more profit then it's supposed to, or the miners can be taken advantage of. In the extreme case, when the real cost of opcode is much higher than the gas pricing, a malicious message sender can send transactions filled with such opcode. As miners tries to execute the transactions, if the execution time were long enough, it is possible that the nodes cannot reach consensus in the protocol. The Ethereum network has experienced such attacks in 2016 and was forced include the EIP-150, bumping the price of SLOAD instruction, as a response. As the cost may change, gas pricing mismatch continues to happen, and "Broken Metre: Attacking Resource Metering in EVM" by Livshits and Perez measures the time needed to execute each opcode and stated that by the current network and opcode pricing, an attacker can craft transactions that requires full nodes to compute around 78 seconds. This would force all the full nodes that runs on a regular computer to be out of sync with budget as low as $148 in USD.

4.5 Reentrancy issues with EIP-1283

EIP-1283 was a proposal that were originally included in the Constantinpole upgrade of the Ethereum Network. The EIP proposed to make the gas price of SSTORE cheaper. As an unexpected affect, this nulled the original intention of guarding reentrancy attack using transfer() or send() in Solidity. Both functions limits the amount of gas that can be used in the fallback function to be 2300 and were able to prevent reentrancy as such, however, this will no longer be the case as the gas price is changed if EIP-1283 were to be included.

5 Gas related EIPs included in Istanbul

The Istanbul upgrade includes gas related EIPs such as EIP-1108, EIP-1884, EIP-2028, and EIP-2200. EIP-2028 proposed to reduce the gas cost of opcode GTXDATANONZERO from 68 gas per byte to 16 gas per byte so that more data can fit into a block. The proposal aims to improve scalability of layer 2 and stateless clients. EIP-2200 considers the SLOAD gas cost change in EIP-1884 and changed the cost for SSTORE accordingly. We would present in the EIP-1108 and EIP-1884 in more detail below.

5.1 EIP-1108

EIP-1108 greatly reduces the price of elliptic curve arithmetic precompiled contracts. The proposal aims to provide better support for privacy and scaling solutions. The new gas price was proposed by measuring computation time on the function ecerecover. Note that one of the reasons that it can be made cheaper is that the underlying libraries of the client implementations were updated.

- ECADD was changed from 500 to 150
- ECMUL was changed from 40000 to 6000
- The price for pairing check were changed from 80 000 * k + 100 000 to 34 000 * k + 45 000.

5.2 EIP-1884

"EIP-1884: Repricing for trie-size-dependent opcodes" aims to address the pricing mismatch due to network state increase. The proposal measured and compared the execution time of each opcode at different blocks and discovered that the increase of Ethereum state has made opcode related to accessing the state trie more expensive.

- SLOAD was changed from 200 to 800
- BALANCE was changed from 400 to 700
- EXTCODEHASH was changed from 400 to 700

Note that the SLOAD opcode here have been repriced in previous update that included EIP-150. EIP-150 have changed the gas price of the SLOAD opcode from 50 to 200 as a response to the DoS attack launched to the network.

6 Security issues on EIP-1884

As EIP-1884 increases the price for some operations, some contracts that has a fallback function that may interact with strict gas stipend (namely, transfer and send) could start failing. For example, the function like below was in the 2300 stipend before the update, but it would start failing. To be specific ConditionA , ConditionB, ConditionC are storage slots and read by the miners. Before the update, the gas cost for these load would be 200 * 3 = 600, however, after the update it would become 800 * 3 = 2400, exceeding the 2300 gas stipend.

```
modifier onlyIfConditionAB
{
  require(ConditionA);
  require(ConditionB);
  _;
}

function () public onlyIfRunning payable {
  require(ConditionC);
  LogEthReceived(msg.sender, msg.value);
}
```

The code that would cause failure are not always easy to spot, as expensive log events could also fail if the log requires reading from the storage as well. For example:

```
function() public payable{
  LogEthReceived(msg.sender, StorageA, StorageB, StorageC);
}
```

The code above would fail because it read from 3 storages. In practice, not all variables are storage and it would require close inspection to spot the failure.

7 Security Implications for developers and potential future works

While EIP-1108, 1884, 2028, 2200 all make gas pricing changes according to the current network, it is unlikely that these would be final. As can be seen above, real cost changes due to several different factors.

- Implementation update: for example, the library update in EIP-1108
- Network change: such as the state changes in EIP-1884
- Hardware advancement: the time to execute would certainly decrease with Moore's law in place.

As gas pricing keeps changing, it may be interesting to explore the idea of dynamic pricing in Ethereum. This is indeed hinted by Vitalik as a mitigation to the problem pointed out in "Network level DoS: Mismatch of opcode gas pricing and real execution cost". To quote directly from Vitalik: "I have considered an approach where the cost of each opcode goes up the more it has been used in the same block, as a generic second line of defense against VM DoS"

For smart contract developers, here's a writeup to avoid gas related issues on Ethereum based on the review and analysis above.

7.1 Avoid unbounded loops

To mitigate "Contract Level DoS: Growing arrays, growing gas consumption", developer should avoid using unbounded loops. If one has to be used, then it is also suggested that the developer have a good understanding on how large could the array grow and estimate the gas consumption in the function.

7.2 Limit gas usage on all transactions, or user should pay

To mitigate the problem "Trick message sender into burning unnecessary gas", when writing an application to interact with the blockchain, the developer should not only check the gas estimation via simulating, but also put a limit on the gas usage to avoid the address to drain your funds by having you to burn gas. If this limit is reached, the application should alert the developer and consider slowing down or pause the execution of similar requests to allow response time.

Though not always desirable for user experience reasons, another way to mitigate the problem is to make sure that the user who initiated the transaction would pay the gas price.

7.3 Use the function 'call' instead of 'transfer' and 'send'

Both functions transfer and send hardcoded the amount of gas that can be used in a fallback function. As we have seen in the previous discussions, gas price will likely change for the coming hard forks and even change between blocks if dynamic pricing is to be introduced. While it has been recommended as a best practice to use transfer and send in the smart contract before, we suggest otherwise. Developers should use call and allow the smart contract to calculate or update the gas limit.

7.4 Reentrancy guard

While Checks-Effects-Interaction pattern and functions transfer and send were suggested as the best practices to guard against reentrancy problem before, we now recommend that one should explicitly use ReentrancyGuard modifiers developed by OpenZeppelin.

8 Conclusion

We see that gas price updates is required to avoid DoS attack in the face of technology advancement, implementation changes, and network state changes. However, in general as smart contract is immutable, changing the underlying mechanism can always break the contracts that was once considered to be secure. As a smart contract developer, it is essential to recognize that unique challenge and monitor the updates of Ethereum even if the smart contract that was deployed was considered secure at deployment time. During development, one should also code accordingly and defensively to be able to react to these changes. The report analyzed the gas related security issues in the past, analyzed the EIPs that is being accepted into the coming Istanbul update, and provides some suggestions to mitigate gas related issues based on the analysis.