Santiago Palladino
Jul 19 · 3 min read

The Parity Wallet Hack Explained

TL;DR

- A vulnerability was found on the Parity Multisig Wallet version 1.5+, that allowed an attacker to steal over 150,000 ETH (~30M USD).
- If you are using the affected wallet contract, make sure to move all funds to a different wallet immediately.
- The OpenZeppelin MultiSig wallet is unaffected by the vulnerability.

. . .

Today, we witnessed the second largest hack, in terms of ETH stolen, in the history of the Ethereum network. As of 12:19 pm UTC, the attacker's account had drained 153,037 ETH from three high-profile multi-signature contracts used to store funds from past token sales. The problem was initially reported by the Parity team, since the affected MultiSig wallet contract was part of the Parity software suite.

As soon as we learned about the vulnerability, we rushed to analyze the cause of the issue, to check if OpenZeppelin's multisignature implementation was affected too.

First and foremost, we want to assure our users that **the OpenZeppelin MultiSig wallet is unaffected** by the vulnerability, and is safe to use.

That being said, we proceed to share our findings, to raise awareness on the pitfalls that made the attack possible.

The attack explained

The attacker sent two transactions to each of the affected contracts: the first to obtain exclusive ownership of the MultiSig, and the second to move all of its funds.

We can see that the first transaction is a call to `initWallet` (line 216 of WalletLibrary):

```
// constructor - just pass on the owner array to the multiowned and  // the limit to daylimit
function initWallet(address[] _owners, uint _required, uint _daylimit) {
  initDaylimit(_daylimit);
  initMultiowned(_owners, _required);
}
```

This function was probably created as a way to extract the wallet's constructor logic into a separate library. This uses a similar idea to the proxy libraries pattern we talked about in the past. The wallet contract forwards all unmatched function calls to the library using `delegatecall` , in line 424 of Wallet:

```
function() payable {
  // just being sent some cash?
  if (msg.value > 0)
    Deposit(msg.sender, msg.value);
  else if (msg.data.length > 0)
    _walletLibrary.delegatecall(msg.data);
}
```

This **causes all public functions from the library to be callable by anyone**, including `initWallet` , which can change the contract's owners. Unfortunately, `initWallet` has no checks to prevent an attacker from calling it after the contract was initialized. The attacker exploited this and simply changed the contract's `m_owners` state variable to a list containing only their address, and requiring just one confirmation to execute any transaction:

```
Function: initWallet(address[] _owners, uint256 _required, uint256 _daylimit) ***

MethodID: 0xe46dcfeb
[0]:0000000000000000000000000000000000000000000000000000000000000060
[1]:0000000000000000000000000000000000000000000000000000000000000000
[2]:0000000000000000000000000000000000000000000000000000000000000060
[3]:0000000000000000000000000000000000000000000000000000000000000001
[4]:000000000000000000000000b3764761e297d6f121e79c32a65829cd1ddb4d32
```

After that, it was just a matter of invoking the execute function to send all funds to an account controlled by the attacker:

```
Function: execute(address _to, uint256 _value, bytes _data) ***

MethodID: 0xb61d27f6
[0]:000000000000000000000000b3764761e297d6f121e79c32a65829cd1ddb4d32
[1]:0000000000000000000000000000000000000000000000116779808c03e4140000
[2]:0000000000000000000000000000000000000000000000000000000000000060
[3]:0000000000000000000000000000000000000000000000000000000000000000
[4]:0000000000000000000000000000000000000000000000000000000000000000
```

This execution was automatically authorized, since the attacker was then the only owner of the multisig, effectively draining the contract of all its funds.

The solution

The attack could have been prevented either by not extracting the constructor logic into the library contract altogether, or better by *not using* `delegatecall` *as a catch-all forwarding mechanism*. The recommended pattern is explicitly defining which library functions can be invoked externally on the wallet contract.

It is important to note that the technique of abstracting logic into a shared library can be quite useful, though. It helps improve code reusability and reduces gas deployment costs. This attack, however, makes clear that **a set of best practices and standards is needed in the Ethereum ecosystem to ensure that these coding patterns are implemented effectively and securely.** Otherwise, the most innocent-looking bug can have disastrous consequences.

At Zeppelin Solutions, we have been working on a solution to these problems, based on our experience building OpenZeppelin and on the many security audits we have performed. We will be sharing more details about this soon.

If you are interested in further discussing the technical details of the attack, and how these security issues can be mitigated, join our slack channel, follow us on Medium, or apply to work with us! We're also available for smart contract security development and auditing work.

Thanks to Francisco Giordano and Manuel Araoz.

Blockchain Ethereum Smart Contracts Security Hacks