Security Considerations

Essential reading to safeguard your contracts After reading this page:

- You will understand the security risks associated with using a Relayer.
- You will learn strategies for safeguarding your contracts effectively.
- You will be exposed to an example of a contract that is susceptible to exploitation. *

Potential Security Risk in Relayer Authentication

A relayer is responsible for dispatching a transaction to an external or public contract method. Given the public nature of this method, it is accessible for invocation by any third party.So, how can we ascertain the legitimacy of the party executing this method?

In certain implementations, we have observed an approach that utilizes a mechanism to ensure that solely the Gelato Relay is authorized to call the contract. This mechanism employs a modifier, known asonlyGelatoRelay . This modifier verifies that the transaction'smsg.sender is theGelatoRelay contract. However, it's crucial to note thatthis form of validation does not offer protection from potential malevolent third parties leveraging the relayer to compromise your contract .

Additional authentication is required to safeguard your contracts!

To ensure robust security for your contracts, additional layers of authentication are indispensable. We urge you to adhere to our tried and tested security best practices. These guidelines have been designed to efficiently manage and mitigate security risks. Let's dive in!

✅ Battle Tested Best Practice

The most prevalent method to authenticate users within the web3 ecosystem relies on the ERC-2771 standard and EIP-712 signature . Gelato offers convenient helper contracts that facilitate the verification and decoding of the user's signature, thereby ensuring that the user initiating the transaction is indeed legitimate.

Gelato's SDK provides two methods that implement theERC-2771 standard behind the scenes:

1. sponsoredCallERC2771
2. callWithSyncFeeERC2771
3.

In both instances, Gelato offers built-in methods to decode themsg.sender andmsg.data , substituting these with_msgSender() and_msgData() , respectively.

We strongly advocate for the use of Gelato's built-in ERC-2771 user signature verification contracts, coupled with theonlyGelatoRelay modifier. This combination offers a robust level of security and helps safeguard your contracts against potential threats.

Addressing the Risk of Relayer Fee Payment

Gelato Relayer can be used in various ways. Among these, two specific methods existcallWithSyncFeeERC2771 andcallWithSyncFee . In these, the target contract is responsible for transferring the fees to thefeeCollector .

Gelato providesRelay Context Contracts , which include helper methods that simplify the extraction process forfeeCollector ,fee , andfeeToken . The fees are transferred by invoking the_transferRelayFee() method.

```

Copy // DANGER! Only use with onlyGelatoRelay or_isGelatoRelay before transferring function_transferRelayFee()internal{ _getFeeToken().transfer(_getFeeCollector(),_getFee()); }

```

The following code sample illustrates howfeeCollector is extracted from thecallData :

```

Copy uint256constant_FEE_COLLECTOR_START=72;// offset: address + address + uint256

// WARNING: Do not use this free fn by itself, always inherit GelatoRelayContext //solhint-disable-next-line func-visibility, private-vars-leading-underscore function_getFeeCollectorRelayContext()purereturns(addressfeeCollector) { assembly{ feeCollector:=shr( 96, calldataload(sub(calldatasize(),_FEE_COLLECTOR_START)) ) } }
```

```
```

This snippet lacks a built-in security check or protective measure; it simply extracts thefeeCollector from thecallData .

Without additional safeguards, this implementation is susceptible to Miner Extractable Value (MEV) front running.

Therefore, any external actor could potentially call the target contract and encode their addresses asfeeCollector .

Given these risks, it is absolutely essential to implement the following security best practices.

✅ Battle Tested Best Practice

Alongside implementing theRelay Context Contracts , it's crucial to verify that themsg.sender of the transaction is theGelatoRelay address before executing fee transfers.

```
```

Copy // Using onlyGelatoRelay modifier functiontargetMethod()externalonlyGelatoRelay{ ... // If you are not using ERC-2771 remember to authenticate all // on-chain relay calls to your contract's methods even if you // identify GelatoRelay as the msg.sender // The following pseudocode signifies an authentication procedure _yourAuthenticationLogic();

// Payment to Gelato _transferRelayFee(); ... }

// Or alternatively using _isGelatoRelay(address _forwarder) method functiontargetMethod()external{ ... // If you are not using ERC-2771 remember to authenticate all // on-chain relay calls to your contract's methods even if you // identify GelatoRelay as the msg.sender // The following pseudocode signifies an authentication procedure _yourAuthenticationLogic();

if(_isGelatoRelay(msg.sender)) { // Payment to Gelato _transferRelayFee(); } ... }

```
```

✅ Additional Security Layer

The aforementioned best practice ensures protection from front-running and unauthorized third-party fund drains. However, if your use case demands heightened control over the fees, you can further minimize risk by introducing amaxFee into your function using the method_transferRelayFeeCapped(uint256 maxFee) .

```
```

Copy // Using onlyGelatoRelay modifier functiontargetMethod()externalonlyGelatoRelay{ ... // If you are not using ERC-2771 remember to authenticate all // on-chain relay calls to your contract's methods even if you // identify GelatoRelay as the msg.sender // The following pseudocode signifies an authentication procedure _yourAuthenticationLogic();

// Payment to Gelato _transferRelayFeeCapped(maxFee); ... }

// Or alternatively using _isGelatoRelay(address _forwarder) method functiontargetMethod()external{ ... // If you are not using ERC-2771 remember to authenticate all // on-chain relay calls to your contract's methods even if you // identify GelatoRelay as the msg.sender // The following pseudocode signifies an authentication procedure _yourAuthenticationLogic();

if(_isGelatoRelay(msg.sender)) { // Payment to Gelato _transferRelayFeeCapped(maxFee); } ... }

```
```

For more detailed information on utilizing_transferRelayFeeCapped(uint256 maxFee) , please consult our comprehensive guidehere .

Example of a Poor and Insecure Implementation

TheVeryDummyWallet in the following example gives the impression of being a well-constructed implementation of the Gelato Relay, since it:

- Inherits theGelatoRelayContext
- Implements theonlyGelatoRelay
- modifier
- Transfers the fees using the built-in method_transferRelayFee()
-

However, this contract has a critical flaw: any user can create a request by calling the Gelato Relay and passing any "to" address to thesendToFriend() method.This contract does not implement any form of user authentication or authorization, making it susceptible to exploitation .

WARNING: THIS IS A BAD EXAMPLE. DO NOT REPLICATE

```

Copy // SPDX-License-Identifier: MIT pragmasolidity0.8.17;

import{ GelatoRelayContext }from"@gelatonetwork/relay-context/contracts/GelatoRelayContext.sol";

import{IERC20}from"@openzeppelin/contracts/token/ERC20/IERC20.sol"; import{ SafeERC20 }from"@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contractVeryDummyWalletisGelatoRelayContext{ // sendToFriend is the target function to call // this function uses this contract's mock ERC-20 balance to send // an _amount of tokens to the _to address. functionsendToFriend( address_token, address_to, uint256_amount )externalonlyGelatoRelay{ // payment to Gelato _transferRelayFee();

// transfer of ERC-20 tokens SafeERC20.safeTransfer(IERC20(_token),_to,_amount); } }

```

Previous What is Relaying? Next ERC-2771 Delegatecall Vulnerability Last updated3 months ago On this page *Potential Security Risk in Relayer Authentication * Addressing the Risk of Relayer Fee Payment* Example of a Poor and Insecure Implementation