Blaize.Security

May 4th, 2023 / V. 1.0

EVERDUES

SMART CONTRACT AUDIT

TABLE OF CONTENTS

AUDIT RATING

SCORE **10** /10

The scope of the project includes EverDues' set of contracts:

contracts

MultiOwnable.sol
RecurringPayments.sol

Repository:

https://github.com/EverDues/evm-smart-contracts

Branch: main

Initial commit:

- e3b5b571539205ef0fc3f84b29b392d7af4c75d4

Final commit:

- 047bad18c740cc4ee2899fc08b590ef3721e727b

TECHNICAL SUMMARY

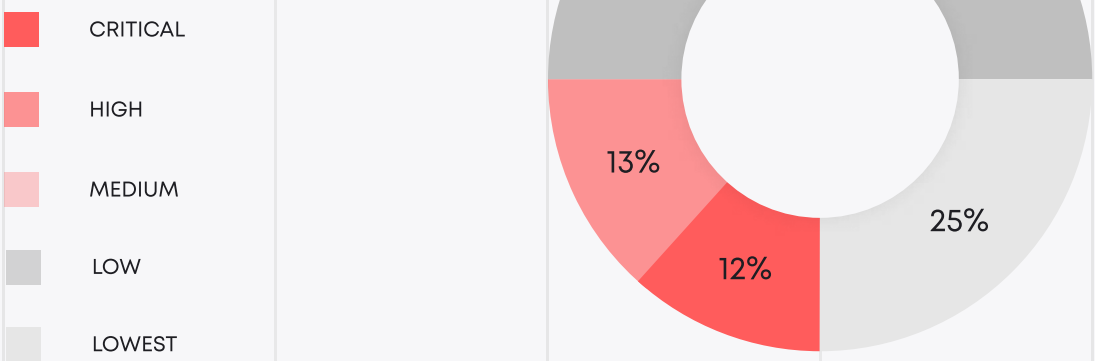
During the audit, we examined the security of smart contracts for the EverDues protocol. Our task was to find and describe any security issues in the smart contracts of the platform. This report presents the findings of the security audit of the **EverDues** smart contracts conducted between April **27th, 2023** and **May 3rd, 2022.**

Testable code

The code is 100% testable, which is above the industry standard of 95%.

The scope of the audit includes the unit test coverage, which is based on the smart contract code, documentation, and requirements presented by the EverDues team. The coverage is calculated based on the set of Hardhat framework tests and scripts from additional testing strategies. However, to ensure the security of the contract, the Blaize.Security team suggests that the EverDues team launch a bug bounty program to encourage further active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of the detected issues and their severity. A total of 8 problems were found. All 8 issues were fixed or verified by the EverDues team.

	FOUND	FIXED/VERIFIED
Critical	1	1
High	1	1
Medium	0	0
Low	4	4
Lowest	2	2

SEVERITY DEFINITION

Critical

The system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

High

The system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge data or financial leak. Requires immediate fixes and a further check.

Medium

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

Low

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

Lowest

The system does not contain any issues critical to the secure work of the system, yet is relevant for best practices.

AUDITING STRATEGY AND TECHNIQUES APPLIED/PROCEDURE

We scanned the smart contracts for commonly known and more specific vulnerabilities:

- Unsafe type inference;
- Timestamp Dependence;
- Reentrancy;
- Implicit visibility level;
- Gas Limit and Loops;
- Transaction-Ordering Dependence;
- Unchecked external call - Unchecked math;
- DoS with Block Gas Limit;
- DoS with (unexpected) Throw;
- Byte array vulnerabilities;
- Malicious libraries;
- Style guide violation;
- ERC20 API violation;
- Uninitialized state/storage/ local variables;
- Compile version not fixed.

Procedure

We checked the contracts for the following parameters:

- Whether the contract is secure;
- Whether the contract corresponds to the documentation;
- Whether the contract meets the best practices in the efficient use of gas, code readability.

Automated analysis:

We scanned the contracts using several publicly available automated analysis tools such as Mythril, Solhint, Slither, and Smartdec. All issues found were verified manually.

Manual audit:

We manually analyzed the smart contracts to identify potential security vulnerabilities. Our analysis involved a comparison of the smart contract logic with the description provided in the documentation.

EXECUTIVE SUMMARY

Blaize Security team has conducted the audit for the EverDues protocol. The protocol represents a platform for recurring payments and subscriptions. It utilizes ERC-20 tokens to pay for subscriptions and the payment is executed automatically. All it needs from the user is to approve tokens to the protocol in advance.

The objective of the audit was to assess the security of smart contracts against the list of common vulnerabilities as well as against the auditors' internal check-list, check that contracts are optimized in terms of gas consumption, and validate the security of users' funds. This includes verifying the protocol can spend only a certain amount of a particular token to the correct destination address in a correct period. From the protocol's perspective, it needed to be validated that users can avoid payments and that fees are properly collected.

The audit discovered one critical, one high, and several low and lowest issues. The critical issue was found in the access control contract, MultiOwnable. The issue occurred because the default admin role of AccessControl.sol was neither granted nor changed to another owner role in the constructor. The EverDues team has successfully fixed this issue by granting a default admin role to the deployer of the contracts. The high issue was connected to the possibility for users to avoid the first payment of the subscription. The issue occurred because the ID of the subscription was generated off-chain without validating the input parameters. Thus, users could have passed invalid parameters while creating a subscription and avoided the first payment while still creating a valid subscription. The EverDues team has also successfully fixed this issue by generating the subscription id on-chain based on input parameters.

Other issues were connected to the lack of validations, usage of custom errors, visibility of variables, and the validation of business logic. The EverDues team has successfully fixed or verified all of the issues.

The overall security of the protocol is high-enough. Contracts are well-written, contain a sufficient natSpec, and have additional documentation. The Blaize Security team carefully checked the flow of subscriptions with additional tests. Once the EverDues team has applied all the fixes, the smart contracts have passed all the security tests. It should also be noted that based on the protocol's logic, one user should have only one subscription. While there are no such restrictions in the smart contracts, the EverDues team has verified that it will be checked in the dApp, and users will only be able to have one valid subscription.

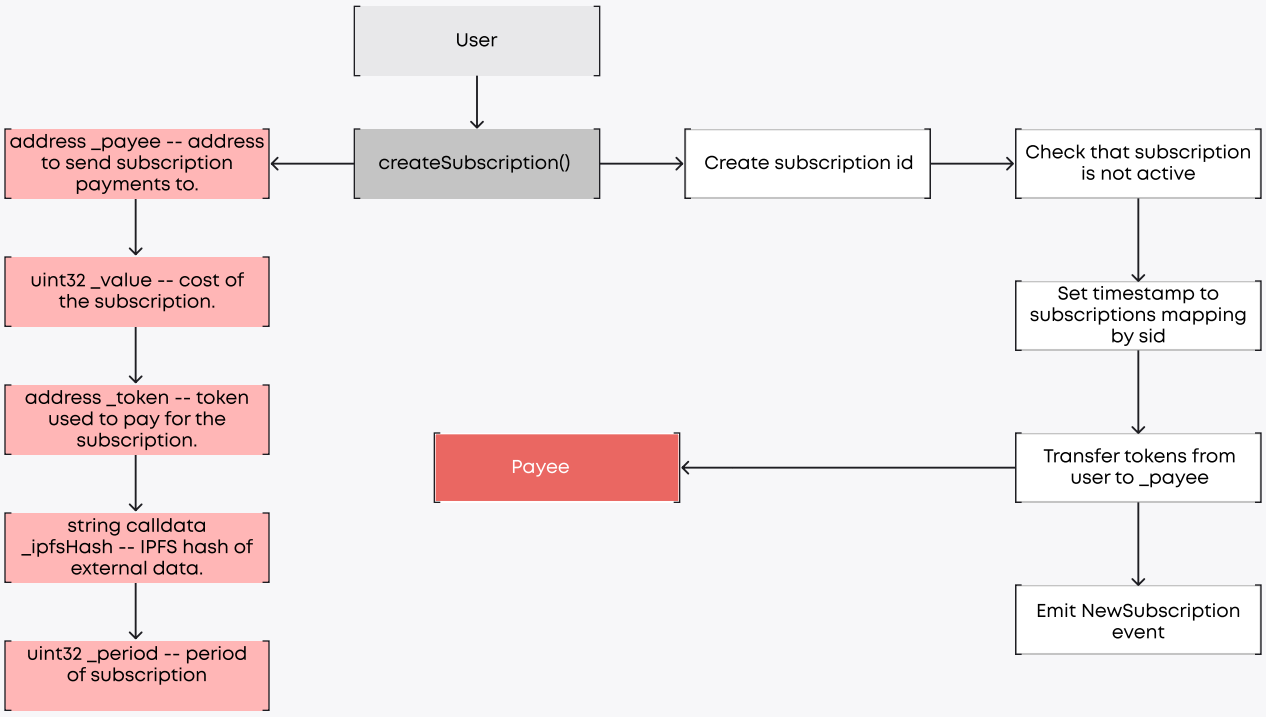
	RATING
Security	9.9
Gas usage and logic optimization	10
Code quality	10
Test coverage	10
Total	10

EVERDUES CONTRACT

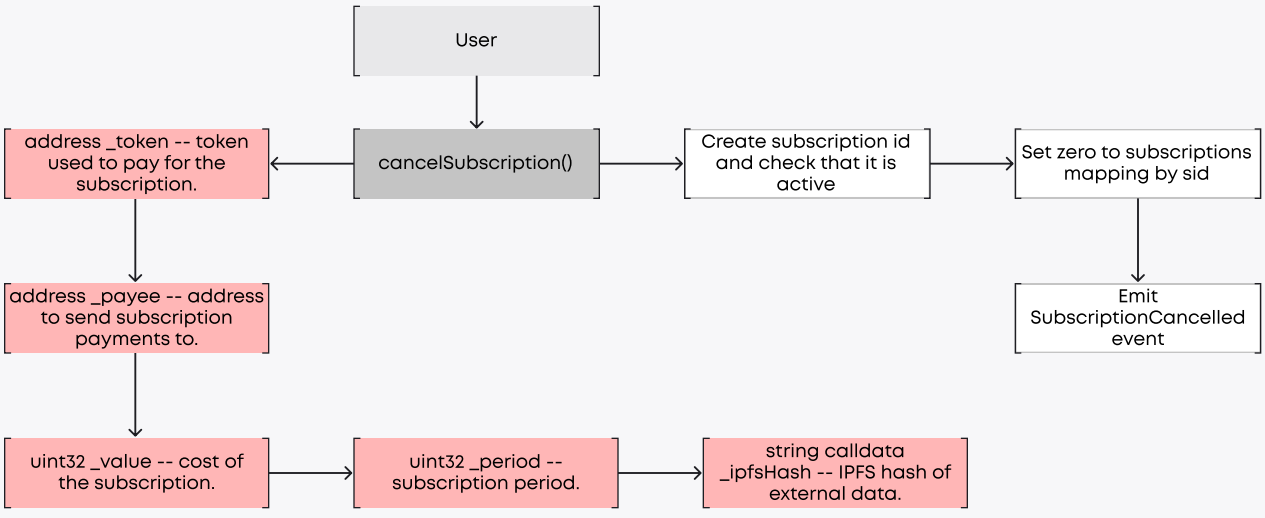
RecurringPayments.sol

EverDues is a protocol that enables users to pay for subscriptions using crypto. Users can create subscriptions, which will then be used to pay for using crypto.

Create subscription



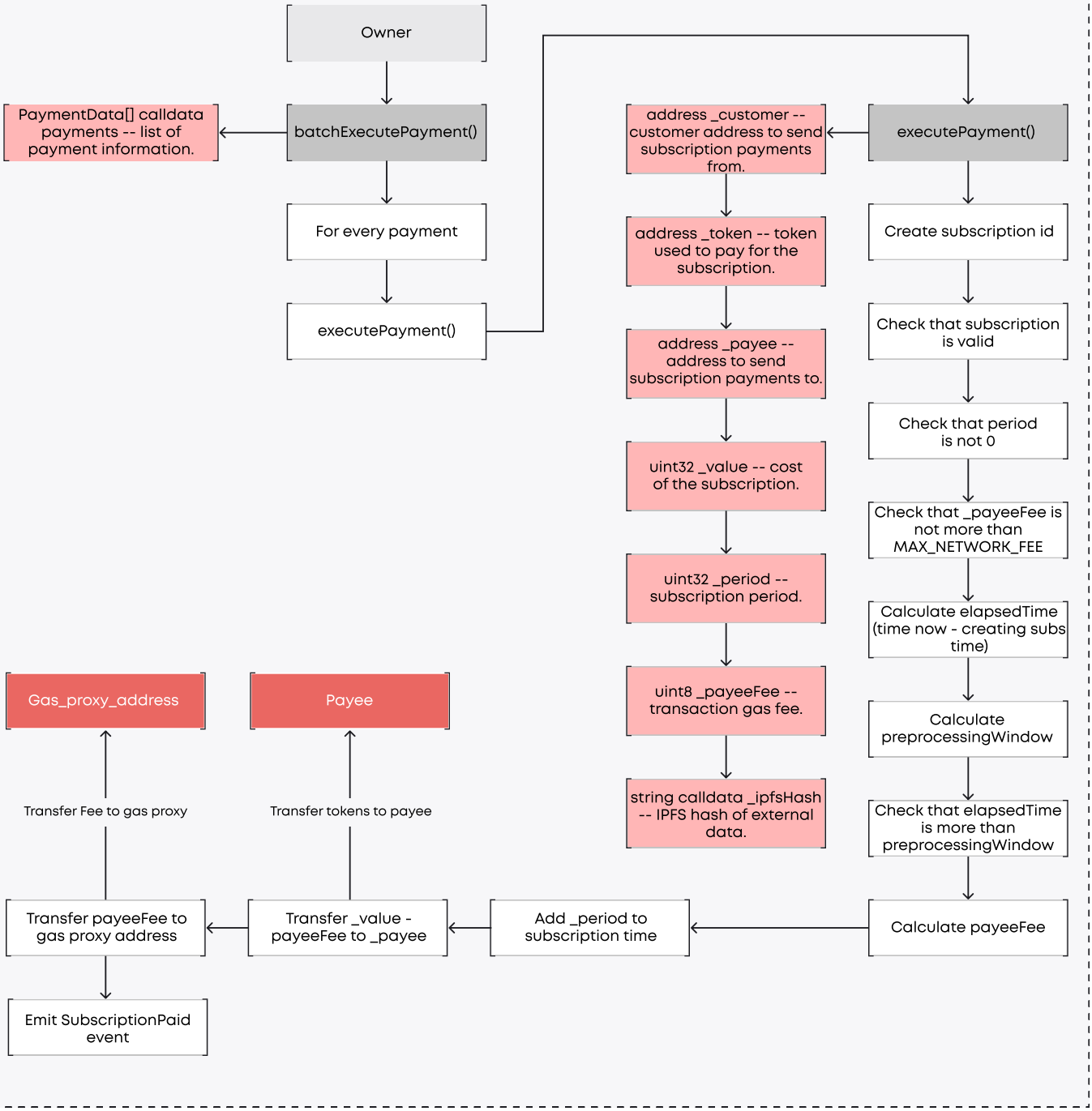
Cancel subscription



EVERDUES CONTRACT

RecurringPayments.sol

Execute payment



COMPLETE ANALYSIS

CRITICAL-1	✔ Resolved

The default admin is not set.

MultiOwnable.sol
In the AccessControl contract from Openzeppelin, the main role by default is DEFAULT_ADMIN_ROLE, which controls other roles. The function grantRole() checks not the case where the sender has the same role but the role that controls granting role. In the MultiOwnable contract addOwner() and removeOwner() functions will revert because no DEFAULT_ADMIN_ROLE was set. In the constructor, default admin should be granted to at least one account, or the admin role should be changed using the _setRoleAdmin() function. This issue is marked as critical since, as for now, only one account has an OWNER_ROLE. However it is unable to set new owners, which doesn't correspond to the logic of smart contracts.

Recommendation:

Grant the DEFAULT_ADMIN_ROLE to msg.sender **OR** set the admin role for the OWNER_ROLE role.

Post-audit:

The default admin is set in the constructor. The addOwner() and removeOwner() functions can only be invoked if the user has both DEFAULT_ADMIN_ROLE and OWNER_ROLE. DEFAULT_ADMIN_ROLE also can avoid checking on owners in the removeOwner() function by calling the revokeRole() function. According to the EverDues team, such functionality is intended as DEFAULT_ADMIN_ROLE is similar to super admin, which can avoid the validation.

HIGH-1	✔ Resolved

The first payment can be performed with a wrong token/wrong quantity/to wrong payee.

RecurringPayments.sol: createSubscription().
Since the hash of ȧid`consists of certain important parameters such as token, payee, and value, it is important to ensure that these parameters are actual when a subscription is created. However, during the subscription creation, parameters `_payee,` `_value,` and `_token`are not validated to be part of ȧid.` This happens since one part of ȧid`is passed as a preprocessed hash `_sid.` As a result, if a malicious actor calls the function directly, he can pass a valid `_sid,` but other parameters will be invalid to avoid the first payment.
Parameter `_ipfsHash` is also not validated. As this parameter is passed to the event NewSubscription which can be essential for the protocol, it is also suggested to validate that it is a part of ȧid.`

Recommendation:

Either form `_sid`inside of the function instead of passing a preprocessed hash **OR** consider restricting the function so that only authorized members of the protocol can call it and ensure that all the parameters are valid (e.g., backend).

Post-audit:

`_sid` is now created inside the function instead of processing it off-chain. Thus, the first payment is performed correctly.

LOW-1	✔ Resolved

Batch executes payment could revert if one subscription is false.

RecurringPayments.sol: batchExecutePayment().
Using batchExecutePayment() could cause a transaction to revert if one subscription does not pass the executePayment() requirement. For example, if 10 payments are passed to the function, 9 payments are valid, and 1 payment is invalid (e.g., canceled), then all 10 payments will not be executed.

Recommendation:

Verify that passed payments are valid without reverting the whole transaction **OR** verify that such logic is intended.

Post-audit:
Since an owner invokes the batch function, every subscription will be checked before function execution.

LOW-2	✔ Resolved

Subscription is not checked when canceling.

RecurringPayments.sol: cancelSubscription().
When a user cancels a subscription, it is not checking if this subscription was created. In this case, it is better to check if the subscription is valid and could be canceled to avoid cases where a user has made a tiny mistake in passed parameters and canceled a wrong subscription, thinking that he canceled the subscription he intended to cancel.

Recommendation:

Check the subscription before canceling.

Post-audit:
The subscription is now checked to be valid before canceling.

LOW-3	✔ Resolved

`gas_proxy_address` is not set in the constructor.

RecurringPayments.sol
The gas proxy address is zero address after the contract deployment. If no changes are made, the executePayment() function will always revert, or fee tokens will be burned (transferred to zero address). To ensure everything will work as expected, variables should be set in the constructor.

Recommendation:

Initialize the gas_proxy_address variable in the constructor of the RecurringPayments contract.

Post-audit:
The gas_proxy_address is now set in the constructor.

LOW-4	✔ Resolved

Variables and constants visibility is not marked explicitly.

RecurringPayments.sol.
By default, all variables have internal visibility when it is not marked explicitly. However, it is recommended to mark the visibility explicitly, even if it has to be private.

Recommendation:

Add visibility to variables.

Post-audit:
The visibility of variables was added.

LOWEST-1	✔ Resolved

Custom errors should be used.

RecurringPayments.sol: createSubscription(), line 62, executePayment(), lines 109-111, 114.
MultiOwnable.sol: onlyOwner(), line 14, removeOwner(), line 23
Starting from the 0.8.4 version of Solidity it is recommended to use custom errors instead of storing error message strings in storage and use “require” statements. Using custom errors is more efficient in terms of gas spending and increases the code readability.

Recommendation:

Use custom errors.

Post-audit:

Custom errors are used now.

LOWEST-2	✔ Verified

Unclear payment/period process.

RecurringPayments.sol

1. It is not fully clear how the user flow in the protocol is working. As for now, it looks like this: The user creates a subscription -> pays for the creation -> the admin executes the user subscription -> the user pays for the subscription again. What will happen if the user wants a different period, should a new subscription be created and the old one be canceled? If the user cancels the subscription before execution, should he get a refund?
2. It is also not yet clear how the protocol handles users' approvals, necessary for the function batchExecutePayment(). It could be the time when the user can revoke his approvals to protocol, and in this case executePayment() function will revert. It is not safe to ask users to approve the maxUint value.

The first part of the issue is not a security issue but rather a validation of user flow. At the same time, the second part can be a problem for the protocol if the user doesn't grant approval in time or a security issue for users if they are asked to grant unlimited allowance to the protocol.

Recommendation:

Clarify the flow of the subscription process and validate how approvals will be granted on the platform.

Post-audit:

The EverDues team has verified that the subscription can't be upgraded by design. For value/period changes, users need to unsubscribe and subscribe to a different plan. This UI/UX is handled on the frontend. Subscribing more than once to the same destination address through the UI is impossible. They stated that the user only must delete and subscribe again.

		contracts\ MultiOwnable.sol RecurringPayments.sol
✓	Re-entrancy	Pass
✓	Access Management Hierarchy	Pass
✓	Arithmetic Over/Under Flows	Pass
✓	Delegatecall Unexpected Ether	Pass
✓	Default Public Visibility	Pass
✓	Hidden Malicious Code	Pass
✓	Entropy Illusion (Lack of Randomness)	Pass
✓	External Contract Referencing	Pass
✓	Short Address/Parameter Attack	Pass
✓	Unchecked CALL Return Values	Pass
✓	Race Conditions/Front Running	Pass
✓	General Denial Of Service (DOS)	Pass
✓	Uninitialized Storage Pointers	Pass
✓	Floating Points and Precision	Pass
✓	Tx.Origin Authentication	Pass
✓	Signatures Replay	Pass
✓	Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY BLAIZE SECURITY TEAM

MultiOwnable

Deployment
- ✓ Sets the deployer as the initial owner when initialization

Add owner
- ✓ Adds a new owner (96ms)
- ✓ Prevents non-owners from adding a new owner (55ms)

Remove owner
- ✓ Removes an owner (47ms)
- ✓ Prevents non-owners from removing an owner
- ✓ Prevents removing the last owner (62ms)

RecurringPayments

Deployment
- ✓ Sets the deployer as the initial owner

createSubscription
- ✓ Creates a new subscription and transfers the subscription cost (103ms)
- ✓ Reverts if an active subscription already exists (73ms)

cancelSubscription
- ✓ Cancels an existing subscription (68ms)
- ✓ Reverts if the subscription does not exist

executePayment
- ✓ Executes a subscription payment and transfers the subscription cost and network fee (87ms)
- ✓ Calculates the preprocessing window correctly (88ms)
- ✓ Reverts if the subscription does not exist or has been cancelled
- ✓ Reverts if the subscription period is zero (lifetime subscription) (58ms)
- ✓ Reverts if the network fee is more than the maximum allowed network fee (63ms)
- ✓ Reverts if the subscription has already been paid for this period (65ms)

encodeSubscriptionId
- ✓ Generates a unique subscription ID based on the given parameters

getSubscriptionTimestamp
- ✓ Retrieves the timestamp of the last payment for a subscription (57ms)

```
    # setGasProxyAddress
  ✓ Sets the gas proxy address
  ✓ Reverts if the caller is not the owner (47ms)
    # batchExecutePayment
  ✓ Reverts if the caller is not the owner (83ms)

    22 passing (4s)
```

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS	% LINES
contracts\	100	100	100	100
MultiOwnable.sol	100	100	100	100
RecurringPayments.sol	100	100	100	100
All files	**100**	**100**	**100**	**100**

DISCLAIMER

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.