shieldify

# Bastion Wallet

SECURITY REVIEW

Date: 1 March 2024

# CONTENTS

## 1. About Shieldify

We are Shieldify Security – Revolutionizing Web3 security. Elevating standards with top-tier reports and a unique subscription-based auditing model. Learn more about us at: shieldify.org or @ShieldifySec

## 2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new prlems and necessitate additional auditing.

## 3. About Bastion Wallet

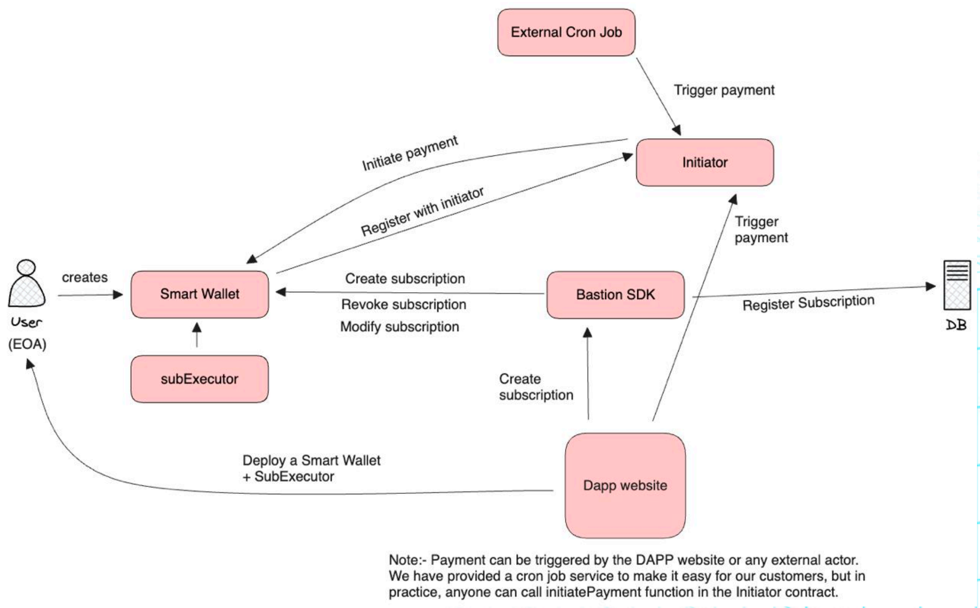
An open-source Account Abstraction SDK enabling seamless integration of multi-chain wallets into Dapps.

Bastion is a modular, lightweight, open-source account abstraction SDK that allows you to integrate decentralized wallet functionality into your applications easily. It is a fully ERC4337-compatible TypeScript SDK, which ensures type safety. It abstracts away blockchain complexity, enabling seamless integration of multi-chain wallets.

With Bastion, users can securely store assets from different blockchains in a unified interface. The SDK handles all blockchain interactions in the background, providing a simplified developer experience.

Learn more about Bastion's concept and the technicalities behind it here.

### 3.1. Observations



External Cron Job
Trigger payment
Initiator
Initiate payment
Register with initiator
Trigger payment
creates
User (EOA)
Smart Wallet
subExecutor
Create subscription
Revoke subscription
Modify subscription
Bastion SDK
Register Subscription
DB
Create subscription
Deploy a Smart Wallet + SubExecutor
Dapp website
Note:- Payment can be triggered by the DAPP website or any external actor. We have provided a cron job service to make it easy for our customers, but in practice, anyone can call initiatePayment function in the Initiator contract.

**Figure 1:** Bastion's Architecture

### 3.2 Privileged Roles and Actors

- Subscription management includes creating, modifying, or revoking a subscription which can only be called by the entry point, owner, or the contract itself.
- Payment processing can only be initiated by the subscription initiator.
- The `Initiator` contract is a singleton contract, deployed for every dApp.

### 3.3 Unacceptable Actions

**1.** `SubExecutor.sol`

- Functions like `createSubscription()`, `modifySubscription()`, and `revokeSubscription()` must not be accessible by unauthorized accounts.
- Creating or modifying a subscription with an amount of 0, or setting invalid intervals and durations should be avoided.
- The contract should not attempt to process payments if there are insufficient ERC20 tokens or Ether in the contract.
- Payments should not be processed if the current time is not within the subscription's valid period.
- The `processPayment()` function is protected by the `nonReentrant` modifier to prevent reentrancy attacks, which is crucial for the security of payment processing.

**2.** `Initiator.sol`

- Functions `withdrawETH()` and `withdrawERC20()` should only be accessible by the contract owner to prevent unauthorized withdrawals of funds.
- The contract should prevent the registration of subscriptions with incorrect or invalid parameters, such as a zero amount or invalid intervals.
- Payments should not be initiated if the subscription is not active, the amount is zero, or the payment interval conditions are not met.

## 4. Risk Classification

| **Severity** | **Impact: High** | **Impact: Medium** | **Impact: Low** |
|---|---|---|---|
| **Likelihood: High** | Critical | High | Medium |
| **Likelihood: Medium** | High | Medium | Low |
| **Likelihood: Low** | Medium | Low | Low |

### 4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

### 4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

## 5. Security Review Summary

The security review lasted 6 days and a total of 144 hours have been spent by the three smart contract security researchers from the Shieldify team. Two additional researchers from Shieldify's newly formed fuzzing team conducted fuzz tests, leveraging Halmos, Echidna and Foundry, and collectively dedicated an additional 112 hours.

Overall, the code is readable and implements foundational good practices. The audit report contributed by identifying several issues of varying severity, impacting improper subscription duration, and insufficient input validation, among other less severe findings.

The contracts in scope do not contain any tests. This, combined with the lack of any documentation required a lot of explanation from the Bastion team, which they were happy to provide.

## 5.1 Protocol Summary

| **Project Name** | **Bastion Wallet** |
|---|---|
| **Repository** | bastion-wallet |
| **Type of Project** | ERC-4337, Account Abstraction SDK |
| **Audit Timeline** | 6 days |
| **Review Commit Hash** | 75b2053670153bc5bc003d1e43ce0e54445a542a |
| **Fixes Review Commit Hash** | 79cddfeb6070140a24a2cb5029faa6c01088ffba |

### 5.2 Scope

The following smart contracts were in the scope of the security review:

| File | nSLOC |
|---|---|
| src/subscriptions/Initiator.sol | 69 |
| src/subscriptions/SubExecutor.sol | 151 |
| src/interfaces/IInitiator.sol | 7 |
| src/subscriptions/ISubExecutor.sol | 26 |
| **Total** | **253** |

## 6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **2**
- **Medium** issues: **3**
- **Low** issues: **5**

| ID | Title | Severity |
|---|---|---|
| [H-01] | Logic Flaw In The Subscription's Handling Functions Execution | High |
| [H-02] | Incorrect Token Transfer in `_processERC20Payment()` Function | High |
| [M-01] | Subscriptions Tokens Distinguish Functionality Is Broken | Medium |
| [M-02] | Registering and Paying a Subscription Will Not Be Possible in a Single UserOperation Batch | Medium |
| [M-03] | Hardcoded Value for `validAfter` Argument Could Prevent Users to Initiate a Subscription at a Later Time | Medium |
| [L-01] | A User Can Register an Unlimited Number of Subscriptions | Low |
| [L-02] | Incomplete Subscriber Removal Logic in `Initiator.sol` Contract | Low |
| [L-03] | Protocol Does Not Work Correctly With Tokens That Do Not Revert On Failed Transfer | Low |
| [L-04] | Using the `transfer()` Function of `address payable` Is Discouraged | Low |
| [L-05] | Wrong Validation Checks | Low |

## 7. Findings

### [H-01] Logic Flaw In The Subscription's Handling Functions Execution

#### Severity

High Risk

#### Description

Important state-changing functions in the `Initiator` contract can currently be called by both EOAs and SCAs. If the executor is an EOA without going through Bastion SDK, then the subscriber will be the address of the EOA account and therefore the following attack vectors are present:

- Subscriptions registered from the `Initiator` contract can not be modified in any way compared to those added from the `SubExecutor` contract.

  `Example:` Let's say that afterward, the user wants to modify his subscription through the SubExecutor's logic by calling `modifySubcsription()` . What would happen then is that because in this same function, the `subscriber` is the `msg.sender` , the subscription, previously created through my EOA account will not be able to be modified, because the struct that will be returned will be the one to which the address of the SCA's SubExecutor is mapped and not the one mapped to users' EOA address.

- The `initiatePayment()` function will revert every time since in the `SubExecutor.processPayment()` there is a check that ensures the `msg.sender` is the creator of the subscription which in this case will be the address of the `Initiator` contract.

```
require(msg.sender == sub.initiator, "Only the initiator can initiate
   payments");
```

- The `removeSubscription()` function executed from an EOA can not delete a subscription created from the `SubExecutor` contract.

All of these problems arise from the scenario where the user is able to call the `Initiator.registerSubscription()` function without first calling `SubExecutor.createSubscription()`.

Expected functions execution flow should be:

1. `SubExecutor.createSubscription()`
2. `Initiator.registerSubscription()`
3. `Initiator.initiatePayment()`
4. `SubExecutor.processPayment()`

## Location of Affected Code

File: src/subscriptions/Initiator.sol

```
/// @notice Registers a new subscription for a subscriber
/// @param _subscriber Address of the subscriber
...
function registerSubscription(address _subscriber, uint256 _amount,
   uint256 _validUntil, uint256 _paymentInterval, address _erc20Token)
   public {
// code
  require(msg.sender == _subscriber, "Only the subscriber can register a
     subscription");

  ISubExecutor.SubStorage memory sub = ISubExecutor.SubStorage({
    amount: _amount,
    validUntil: _validUntil,
    validAfter: block.timestamp,
    paymentInterval: _paymentInterval,
    subscriber: _subscriber,
    initiator: address(this),
    erc20TokensValid: _erc20Token == address(0) ? false : true,
    erc20Token: _erc20Token
  });
// code
}
```

```
function removeSubscription(address _subscriber) public {
  require(msg.sender == _subscriber, "Only the subscriber can remove a
     subscription");
  delete subscriptionBySubscriber[_subscriber];
}

function initiatePayment(address _subscriber) public nonReentrant {
// code
  ISubExecutor(subscription.subscriber).processPayment();
}
```

File: src/subscriptions/SubExecutor.sol#L120

```
function processPayment() external nonReentrant {
// code
  require(msg.sender == sub.initiator, "Only the initiator can initiate
     payments");
// code
}
```

### Recommendation

To address this vulnerability, it is crucial to have the core logic and validation in the `Initiator.sol` contract and restrict only the `SubExecutor.sol` contract to call `registerSubscription()`, `removeSubscription()` and `initiatePayment()` functions.

### Team Response

Acknowledged and fixed as suggested.

## [H-02] Incorrect Token Transfer in `_processERC20Payment()` Function

### Severity

High Risk

### Description

The current design of the payment management within the `SubExecutor` contract introduces significant risks of logic failures in token transfer operations, potentially resulting in the inability to fulfill certain subscriptions and the potential locking of tokens.

The **SubExecutor** contract is designed to manage subscriptions and process automatic payments between accounts, using native/ERC20 tokens as a payment method. The current implementation presents an inconsistency in the authorization flow and execution of payments, specifically in the `_processERC20Payment()` function, which attempts to transfer ERC20 tokens from the `SubExecutor` contract to the `initiator` of the subscription using the `transferFrom()` method.

The payment functionality consists of the following steps:

**1. Payment Processing** `processPayment()`**:**

- This function is invoked by the **initiator** to process a payment based on an active subscription. The logic ensures that only the initiator can execute the payment, which is correct and desirable to control the flow of funds. It executes either `_processERC20Payment()` or `_processNativePayment()` whether it's a native payment or ERC20.

**2. ERC20 Token Transfer** `_processERC20Payment()`**:**

- The use of `transferFrom()` implies that the initiator has permitted the `SubExecutor` to withdraw tokens on their behalf, which does not align with the subscription model where the `SubExecutor` is the holder of the funds and must send them directly to the `initiator`. This approach misinterprets the authorization dynamics in ERC20 contracts, leading to a potential failure in payment execution due to "insufficient allowance".

### Location of Affected Code

File: src/subscriptions/SubExecutor.sol#L160

```
/// @notice Processes an ERC20 payment for the subscription
function _processERC20Payment(SubStorage storage sub) internal {
  IERC20 token = IERC20(sub.erc20Token);
  uint256 balance = token.balanceOf(address(this));
  require(balance >= sub.amount, "Insufficient token balance");
  token.transferFrom(address(this), sub.initiator, sub.amount);
}
```

### Proof of Concept

```
function test_FuzzERC20Payment(uint256 tokenBalance, uint256
    paymentAmount) public {
// Ensure reasonable input values.
  vm.assume(tokenBalance >= 1 ether && tokenBalance <= 1000 ether);
  vm.assume(paymentAmount >= 1 wei && paymentAmount <= tokenBalance);

// Mint tokens to SubExecutor and set the allowance for Initiator.
  vm.startPrank(deployer);
  token.mint(address(subExecutor), tokenBalance);
  vm.stopPrank();

// SubExecutor approves Initiator to spend tokens on its behalf.
  vm.startPrank(address(subExecutor));
  token.approve(address(initiator), tokenBalance);
  vm.stopPrank();
// Save initial balances of SubExecutor and Initiator for comparison
   later.
  uint256 subExecutorBalanceBefore = token.balanceOf(address(subExecutor)
     );
  uint256 initiatorBalanceBefore = token.balanceOf(address(initiator));

// Set up a subscription in SubExecutor for the Initiator.
  address owner = subExecutor.getOwner();
  uint256 validAfter = block.timestamp;
  uint256 validUntil = block.timestamp + 90 days;

  vm.startPrank(owner);
  subExecutor.createSubscription(address(initiator), paymentAmount, 30
     days, validUntil, address(token));
  vm.stopPrank();

// Warp time to meet the payment interval.
  vm.warp(validAfter + 30 days + 1);

// Process the payment as the Initiator.
  vm.prank(address(initiator));
  subExecutor.processPayment();

// Verify balances after payment.
  uint256 subExecutorBalanceAfter = token.balanceOf(address(subExecutor))
     ;
  uint256 initiatorBalanceAfter = token.balanceOf(address(initiator));


// Assertions to verify the correct transfer of tokens.
  assertEq(subExecutorBalanceAfter, subExecutorBalanceBefore -
     paymentAmount, "SubExecutor's balance should decrease by the payment
      amount.");
  assertEq(initiatorBalanceAfter, initiatorBalanceBefore + paymentAmount,
      "Initiator's balance should increase by the payment amount.");
}
```

Currently, the test above reverts with `ERC20: insufficient allowance`.

```
Logs:
  ERC20 Token balance of SubExecutor: 3617155993734787769
  Allowance for Initiator to spend SubExecutor's tokens:
     3617155993734787769
```

### Recommendation

Consider changing the code in the following way:

```diff
- token.transferFrom(address(this), sub.initiator, sub.amount);
+ token.transfer(sub.initiator, sub.amount);
```

### Team Response

Acknowledged and fixed as suggested.

## [M-01] Subscriptions Tokens Distinguish Functionality Is Broken

### Severity

Medium Risk

### Description

The function `_processNativePayment()` will be called only in case the `sub.erc20TokensValid` is `false`, and this condition will only be met if the input `_erc20Token` address is set to `address(0)`.

Therefore, the user could add a different token other than ERC20, such as an address of a malicious contract or any other standard. This could lead to unexpected behavior.

### Location of Affected Code

File: src/subscriptions/SubExecutor.sol

```
/// @notice Creates a subscription
/// @param _initiator Address of the initiator
/// @param _amount Amount to be subscribed
/// @param _interval Interval of payments in seconds
/// @param _validUntil Expiration timestamp of the subscription
/// @param _erc20Token Address of the ERC20 token for payment
function createSubscription( address _initiator, uint256 _amount, uint256
    _interval, uint256 _validUntil, address _erc20Token ) external
   onlyFromEntryPointOrOwnerOrSelf {
// code
  getKernelStorage().subscriptions[_initiator] = SubStorage({
    amount: _amount,
    validUntil: _validUntil,
    validAfter: block.timestamp,
    paymentInterval: _interval,
    subscriber: address(this),
    initiator: _initiator,
    erc20Token: _erc20Token,
>>  erc20TokensValid: _erc20Token == address(0) ? false : true
  });
// code
}
```

```
/// @notice Processes a payment for the subscription
function processPayment() external nonReentrant {
// code
//Check whether it's a native payment or ERC20 or ERC721
  if (sub.erc20TokensValid) {
    _processERC20Payment(sub);
  } else {
    _processNativePayment(sub);
  }
// code
}

/// @notice Processes an ERC20 payment for the subscription
function _processERC20Payment(SubStorage storage sub) internal {
  IERC20 token = IERC20(sub.erc20Token);
  uint256 balance = token.balanceOf(address(this));
  require(balance >= sub.amount, "Insufficient token balance");
  token.transferFrom(address(this), sub.initiator, sub.amount);
}
```

### Proof of Concept

**Objective:** Usage of fuzzing to verify the contract's behavior when attempting to register a non-ERC20 token.

**Procedure:**

- Random values are generated for `amount`, `validUntil`, and `paymentInterval`.
- The test attempts to register a subscription with a non-ERC20 token (FalseToken).
- A revert is expected, indicating that the function should not accept a non-ERC20 token.

**Result:**

- The test confirms the contract rejects the registration with a non-ERC20 token, as indicated by the expected revert.

```
function test_primer_failTokenFalse(
  uint256 amount,
  uint256 _validUntil,
  uint256 paymentInterval,
  address FalseToken
) public {
  address subscriber = holders[0];
  amount = bound(amount, 1 ether, 1000 ether);
  paymentInterval = bound(paymentInterval, 1 days, 365 days);
  uint256 validUntil = block.timestamp + bound(_validUntil, 1 days, 365
     days);
```

```
    vm.assume(FalseToken != address(token));
    vm.assume(amount > 0 && paymentInterval > 0 && validUntil > block.
        timestamp);

    vm.prank(subscriber);
    vm.expectRevert();
    initiator.registerSubscription(subscriber, amount, validUntil,
        paymentInterval, FalseToken);
}
```

### Recommendation

Consider creating a requirement that clearly distinguishes between valid tokens, other types of tokens, and invalid tokens. A potential workaround would have been to add a `supportsInterface` call, but most ERC-20 tokens do not implement ERC165, making it unreliable.

Since the `Initiator.sol` contract is a singleton for every dApp, we propose adding a public mapping, containing the addresses of all ERC-20 tokens, that the dApp will be willing to accept as payment. There would also be `whitelistTokenForPayment()` and `removeTokenForPayment()` functions, callable only by the deployer of the `Initiator` contract. If the address of the token exists in the mapping, `_processERC20Payment()` will be called and `_processNativePayment()` otherwise.

This way, users could use valid tokens for payment, avoiding registering malicious tokens, any address, or any standards different from the accepted ones.

- File: `Initiator.sol`

```
+ mapping(address => bool) public whitelistedAddresses;

+ event AddressAdded(address indexed _address);
+ event AddressRemoved(address indexed _address);

+ function whitelistTokenForPayment(address token) external onlyOwner
    {
+   require(!whitelistedAddresses[_address], "Address is already
    whitelisted");
+   whitelistedAddresses[_address] = true;

+   emit AddressAdded(_address);
+ }

+ function removeTokenForPayment(address _address) external {
+   require(whitelistedAddresses[_address], "Address is not
    whitelisted");
+   delete whitelistedAddresses[_address];

+   emit AddressRemoved(_address);
+ }
```

- File: `SubExecutor.sol`

```
/// @notice Processes a payment for the subscription
function processPayment() external nonReentrant {
// code
//Check whether it's a native payment or ERC20 or ERC721
- if (sub.erc20TokensValid) {
+ if (Initiator(_initiator).whitelistedAddresses[sub.erc20Token]) {
    _processERC20Payment(sub);
  } else {
    _processNativePayment(sub);
  }
// code
}
```

### Team Response

Acknowledged and fixed as suggested.

## [M-02] Registering and Paying a Subscription Will Not Be Possible in a Single UserOperation Batch

### Severity

Medium Risk

### Description

The Bastion wallet supports batched userOperations. However, it will not be possible to register a subscription and initiate payment for it in the same batch. That is because with the current implementation, `registerSubscription()` sets v `alidAfter` to `block.timestamp` whereas, the `initiatePayment()` checks if `validAfter < block.timestamp`. These circumstances make both functions non-callable in the same block.

### Location of Affected Code

File: src/subscriptions/Initiator.sol

```
function registerSubscription(
  address _subscriber,
  uint256 _amount,
  uint256 _validUntil,
  uint256 _paymentInterval,
  address _erc20Token
) public {

function initiatePayment(address _subscriber) public nonReentrant {
```

### Recommendation

Ensure that in `registerSubscription()` the `validAfter` parameter is not hard coded but passed as input and is greater or equal to `block.timestamp`, the same applies to the `initiatePayment()`

as well:

```
function registerSubscription(
  address _subscriber,
  uint256 _amount,
  uint256 _validUntil,
+ uint256 _validAfter
  uint256 _paymentInterval,
  address _erc20Token
) public {
// code
+ require(_validAfter >= block.timestamp, "Sub cannot be valid after a
   time in the past")
// code
- validAfter: block.timestamp,
+ validAfter: _validAfter,
// code
function initiatePayment(address _subscriber) public nonReentrant {
  ISubExecutor.SubStorage storage subscription = subscriptionBySubscriber
     [
    _subscriber
  ];
- require(subscription.validAfter < block.timestamp, "Subscription is not
    active");
+ require(subscription.validAfter <= block.timestamp, "Subscription is
   not active");
// code
}
```

### Team Response

Acknowledged and fixed as suggested.

## [M-03] Hardcoded Value for `validAfter` Argument Could Prevent Users to Initiate a Subscription at a Later Time

### Severity

Medium Risk

### Description

In the `registerSubscription()` function the `validAfter` parameter is hardcoded to `block.timestamp`, which would not allow a user to create a subscription, starting at a certain time in the future.

### Location of Affected Code

File: src/subscriptions/Initiator.sol#L33

```
/// @notice Registers a new subscription for a subscriber
/// @param _subscriber Address of the subscriber
/// @param _amount The amount for the subscription
/// @param _validUntil The timestamp until which the subscription is
   valid
/// @param _paymentInterval The interval at which payments should be made
/// @param _erc20Token The ERC20 token address used for payment (address
   (0) for ETH)
function registerSubscription(
  address _subscriber,
  uint256 _amount,
  uint256 _validUntil,
  uint256 _paymentInterval,
  address _erc20Token
) public {
  require(_amount > 0, "Subscription amount is 0");
  require(_paymentInterval > 0, "Payment interval is 0");
  require(msg.sender == _subscriber, "Only the subscriber can register a
     subscription");

  ISubExecutor.SubStorage memory sub = ISubExecutor.SubStorage({
    amount: _amount,
    validUntil: _validUntil,
>>  validAfter: block.timestamp,
    paymentInterval: _paymentInterval,
    subscriber: _subscriber,
    initiator: address(this),
    erc20TokensValid: _erc20Token == address(0) ? false : true,
    erc20Token: _erc20Token
  });
  subscriptionBySubscriber[_subscriber] = sub;
  subscribers.push(_subscriber);
}
```

### Recommendation

Consider introducing an input argument called `_validAfter` and passing it for `validAfter`, including a check that the input value is larger or equal to the `block.timestamp`:

```
function registerSubscription(
  address _subscriber,
  uint256 _amount,
  uint256 _validUntil,
+ uint256 _validAfter
  uint256 _paymentInterval,
  address _erc20Token
) public {
// code
```

```
+ require(_validAfter >= block.timestamp, "Sub cannot be valid after a
   time in the past")
// code
- validAfter: block.timestamp,
+ validAfter: _validAfter,
```

### Team Response

Acknowledged and fixed as suggested.

## [L-01] A User Can Register an Unlimited Number of Subscriptions

### Severity

Low Risk

### Description

A user can register an infinite number of times since there are no restrictions in place to prevent it. It is therefore possible to register a large number of subscriptions such that `registerSubscription()` and `getSubscribers()` functions will run out of gas causing a Denial of Service.

### Location of Affected Code

File: src/subscriptions/Initiator.sol#L13-L42

```
function registerSubscription(address _subscriber, uint256 _amount,
   uint256 _validUntil, uint256 _paymentInterval, address _erc20Token)
   public {
  require(_amount > 0, "Subscription amount is 0");
  require(_paymentInterval > 0, "Payment interval is 0");
  require(msg.sender == _subscriber, "Only the subscriber can register a
     subscription");

  ISubExecutor.SubStorage memory sub = ISubExecutor.SubStorage({
    amount: _amount,
    validUntil: _validUntil,
    validAfter: block.timestamp,
    paymentInterval: _paymentInterval,
    subscriber: _subscriber,
    initiator: address(this),
    erc20TokensValid: _erc20Token == address(0) ? false : true,
    erc20Token: _erc20Token
  });
  subscriptionBySubscriber[_subscriber] = sub;
  subscribers.push(_subscriber);
}
```

### Proof of Concept

#### Objective:

To assess whether the `Initiator.sol` contract allows multiple registrations of the same subscriber and to ensure that no duplicates are added to the subscriber's array.

#### Procedure:

- An attempt is made to register the same subscriber ( `holders[0] / _subscriber` ) five times with random parameters.
- The test verifies that only one registration per subscriber is allowed and that there are no duplicates in the subscriber's array.

#### Expected Results:

- The test should confirm that the contract does not admit multiple registrations for the same subscriber.
- If the test fails, it indicates that the contract allows multiple registrations, which is an undesirable behavior.

#### Conclusion:

This test is crucial for verifying the correct management and data integrity in the `Initiator` contract, ensuring that each subscriber is registered only once and maintaining uniqueness in the list of subscribers.

```
function testFuzzxcMultipleSubscriptions() public {
  address subscriber = holders[0];
  uint256 iterations = 5;

  for (uint256 i = 0; i < iterations; i++) {

    uint256 amount = uint256(keccak256(abi.encodePacked(block.timestamp,
       subscriber, i))) % 100 ether;
    uint256 validUntil = block.timestamp + (1 days + i * 1 days);
    uint256 paymentInterval = (1 days + i * 1 hours);

    vm.prank(subscriber);
    initiator.registerSubscription(subscriber, amount, validUntil,
       paymentInterval, address(token));
    console.log("subscriber:", subscriber);
    console.log("amount:", amount);
    console.log("validUntil:", validUntil);
  }

  address[] memory registeredSubscribers = initiator.getSubscribers();
  assertTrue(registeredSubscribers.length <= 1, "Should not allow
     multiple registrations for the same subscriber");

  for (uint256 i = 1; i < registeredSubscribers.length; i++) {
    console.log("registeredSubscribers[i - 1]:", registeredSubscribers[i
       - 1]);
    console.log("registeredSubscribers[i]:", registeredSubscribers[i]);
    assertTrue(registeredSubscribers[i - 1] != registeredSubscribers[i],
       "No duplicate subscribers should exist");
  }
}
```

### Recommendation

Implement a requirement to ensure that the same user has not registered other subscriptions previously or consider the possibility of adding an upper bound limit on the maximum number of subscriptions.

Sample implementation:

```
+ mapping(address => bool) private subscriberStatus;

function registerSubscription(address _subscriber, uint256 _amount,
   uint256 _validUntil, uint256 _paymentInterval, address _erc20Token)
   public {
  require(_amount > 0, "Subscription amount is 0");
  require(_paymentInterval > 0, "Payment interval is 0");
  require(msg.sender == _subscriber, "Only the subscriber can register a
     subscription");
+ require(!subscriberStatus[_subscriber], "Subscriber already registered
   ");
  // code
  subscriptionBySubscriber[_subscriber] = sub;
+ subscriberStatus[_subscriber] = true;
}
```

```
function removeSubscription(address _subscriber) public {
  require(msg.sender == _subscriber, "Only the subscriber can remove a
     subscription");
+ require(subscriberStatus[_subscriber], "Subscriber not registered");

  delete subscriptionBySubscriber[_subscriber];
+ subscriberStatus[_subscriber] = false;
}

function initiatePayment(address _subscriber) public nonReentrant {
// code
+ require(subscriberStatus[_subscriber], "Subscriber not registered");
// code
}
```

### Team Response

Acknowledged and fixed as suggested.

## [L-02] Incomplete Subscriber Removal Logic in `Initiator.sol` Contract

### Severity

Low Risk

### Description

Registering a new subscription involves updating the `subscriptionBySubscriber` mapping and appending the subscriber's address to the `subscribers` array. However, the `removeSubscription()` method solely removes the subscriber's address from the mapping, leaving potentially invalid subscriptions untouched within the storage array. This oversight, coupled with the absence of a subscription count limit, poses a vulnerability that could be exploited to initiate a Denial of Service (DoS) attack.

### Location of Affected Code

File: src/subscriptions/Initiator.sol#L46

```
/// @notice Removes a subscription for a subscriber
/// @param _subscriber Address of the subscriber
function removeSubscription(address _subscriber) public {
  require(msg.sender == _subscriber, "Only the subscriber can remove a
     subscription");
  delete subscriptionBySubscriber[_subscriber];
}
```

### Proof of Concept

```
function testRemoveSubscription() public {
  address subscriber = address(1);
  uint256 amount = 1 ether;
  uint256 validUntil = block.timestamp + 30 days;
  uint256 paymentInterval = 10 days;

  vm.prank(subscriber);
  initiator.registerSubscription(subscriber, amount, validUntil,
     paymentInterval, address(token));
  vm.prank(subscriber);
  initiator.removeSubscription(subscriber);

  ISubExecutor.SubStorage memory sub = initiator.getSubscription(
     subscriber);
  assertEq(sub.subscriber, address(0));

  address[] memory registeredSubscribers = initiator.getSubscribers();
  bool isSubscriberPresent = false;

  for (uint i = 0; i < registeredSubscribers.length; i++) {
    if (registeredSubscribers[i] == subscriber) {
      isSubscriberPresent = true;
      break;
    }
  }

  assertFalse(isSubscriberPresent, "Subscriber should be removed from the
      subscriber's array");
}
```

Currently, the test above reverts with `Subscriber should be removed from the subscriber's array`
.

```
Logs:
  Error: Subscriber should be removed from the subscriber's array
  Error: Assertion Failed
```

### Recommendation

Modify the `removeSubscription()` function to remove from the `subscribers` array and additionally, consider introducing a subscriptions count limit to prevent potential Denial of Service (DoS) attack.

- File: Initiator.sol

### Recommendation

To address this vulnerability, change the code as follows:

- File: Initiator.sol

```
+ mapping(address => bool) public isSubscriber;
+ mapping(address => uint256) public subscriberByIndex;

function registerSubscription(
  address _subscriber,
  uint256 _amount,
  uint256 _validUntil,
  uint256 _validAfter,
  uint256 _paymentInterval,
  address _erc20Token
) public {
// code

+ if (!isSubscriber[_subscriber]) {
+   subscribers.push(_subscriber);
+   isSubscriber[_subscriber] = true;
+   subscriberByIndex[_subscriber] = subscribers.length - 1;
+ }
}

function removeSubscription(address _subscriber) public {
  require(msg.sender == _subscriber, "Only the subscriber can remove a
     subscription");

  delete subscriptionBySubscriber[_subscriber];

+ uint256 index = subscriberByIndex[_subscriber];
+ address lastSubscriber = subscribers[subscribers.length - 1];
+ subscribers[index] = lastSubscriber;
+ subscriberByIndex[lastSubscriber] = index;
+ subscribers.pop();
}
```

### Team Response

Acknowledged and fixed as suggested.

## [L-03] Protocol Does Not Work Correctly With Tokens That Do Not Revert On Failed Transfer

### Severity

Low Risk

### Description

Some tokens do not implement the ERC20 standard properly but are still accepted by most code that accepts ERC20 tokens. For example, Tether (USDT)'s `transfer()` and `transferFrom()` functions on L1 do not return booleans as the specification requires and instead have no return value. With the current code, if such a call fails but does not revert it will result in inaccurate calculations or funds stuck in the protocol.

### Location of Affected Code

File: src/subscriptions/Initiator.sol#L89

```
/// @notice Withdraws all of a specific ERC20 token held by the contract
    to the owner's address
/// @param _token The ERC20 token address
/// @dev This function can only be called by the contract owner
function withdrawERC20(address _token) public onlyOwner {
  IERC20 token = IERC20(_token);
  token.transfer(owner(), token.balanceOf(address(this)));
}
```

File: src/subscriptions/SubExecutor.sol#L160

```
/// @notice Processes an ERC20 payment for the subscription
function _processERC20Payment(SubStorage storage sub) internal {
  IERC20 token = IERC20(sub.erc20Token);
  uint256 balance = token.balanceOf(address(this));
  require(balance >= sub.amount, "Insufficient token balance");
  token.transferFrom(address(this), sub.initiator, sub.amount);
}
```

### Recommendation

Use OpenZeppelin's `SafeERC20` library and its safe methods for ERC20 transfers.

### Team Response

Acknowledged and fixed as suggested.

## [L-04] Using the `transfer()` Function of `address payable` Is Discouraged

### Severity

Low Risk

### Description

The `transfer()` function only allows the recipient to use `2300` gas. If the recipient uses more than that, transfers will fail. This could, for example, be the case if `initiator` is the address of a Multisig or payment splitter that is supposed to execute additional logic after the withdrawal. Furthermore, gas costs might change in the future, increasing the likelihood of that happening. Also, notice that once the `initiator` address is set to a specific subscription, it could be changed via the `modifySubscription()` function.

Consider the following scenario:

1. During the creation of a subscription, the executor is not aware of this " `transfer() issue` " and sets the `initiator` to a contract address (e.g., a payment splitter) that consumes more than `2300` gas.
2. The user calls `processPayment()` function for a native subscription payment but the execution reverts because the `initiator` consumes more than `2300` gas when receiving the funds.

### Location of Affected Code

File: src/subscriptions/SubExecutor.sol#L166

```
/// @notice Processes a native payment for the subscription
function _processNativePayment(SubStorage storage sub) internal {
  require(address(this).balance >= sub.amount, "Insufficient Ether
     balance");
  payable(sub.initiator).transfer(sub.amount);
}
```

File: src/subscriptions/Initiator.sol#L81

```
/// @notice Withdraws all Ether held by the contract to the owner's
   address
/// @dev This function can only be called by the contract owner
function withdrawETH() public onlyOwner {
  payable(owner()).transfer(address(this).balance);
}
```

### Recommendation

Use `call()` instead of `transfer()` in `_processNativePayment()` and `withdrawETH()` functions:

- File: SubExecutor.sol

```
- payable(sub.initiator).transfer(sub.amount);

+ (bool success, ) = sub.initiator.call{value: sub.amount}("");
+ require(success, "ProcessNativePayment failed.");
```

- File: Initiator.sol

```
- payable(owner()).transfer(address(this).balance);

+ (bool success, ) = owner().call{value: address(this).balance}("");
+ require(success, "WithdrawETH failed.");
```

### Team Response

Acknowledged and fixed as suggested.

## [L-05] Wrong Validation Checks

### Severity

Low Risk

### Description

- The `processPayment()` performs validation checks for `validAfter` and `validUntil` properties but these checks are already performed in the `initiatePayment()` function that executes `processPayment()`.
- When registering a subscription `_validUntil` should be enforced to be larger than `_validAfter` which is set to `block.timestamp`, to ensure that the newly created subscription is valid.
- When registering a subscription in the `createSubscription()` function in the `SubExecutor.sol` contract there is a zero-address check for the `amount` but this check is already performed in the `registerSubscription()` function in `Initiator.sol` contract which is executed internally.

### Location of Affected Code

File: src/subscriptions/Initiator.sol#L70-L71

```
/// @notice Initiates a payment for a given subscriber
/// @param _subscriber Address of the subscriber
/// @dev This function ensures that the subscription is active and the
   payment interval has been reached
function initiatePayment(address _subscriber) public nonReentrant {
// code
  require(subscription.validUntil > block.timestamp, "Subscription is not
      active");
  require(subscription.validAfter < block.timestamp, "Subscription is not
      active");
// code
}
```

File: src/subscriptions/Initiator.sol#L19

```
/// @param _validUntil The timestamp until which the subscription is
   valid
function registerSubscription(address _subscriber, uint256 _amount,
   uint256 _validUntil, uint256 _paymentInterval, address _erc20Token)
   public {
```

File: src/subscriptions/SubExecutor.sol#L48

```
require(_amount > 0, "Subscription amount is 0");
```

### Recommendation

- Consider removing the duplicated validation checks:
  - File: `Initiator.sol`

```
function initiatePayment(address _subscriber) public nonReentrant
    {
// code
- require(subscription.validUntil > block.timestamp, "
   Subscription is not active");
- require(subscription.validAfter < block.timestamp, "
   Subscription is not active");
// code
}
```

  - File: `SubExecutor.sol`

```
- require(_amount > 0, "Subscription amount is 0");
```

- Add the following validation check in the `registerSubscription()` function, and consider setting a minimum subscription's validity period as well:
  - File: `Initiator.sol`

```
+ require(_validUntil > _validAfter, "Wrong subscription's
   timestamp validity");
```

### Team Response

Acknowledged and fixed as suggested.

our shielding · Your smart contracts, our shielding · Your smart

shieldify

# Thank you!