1. A User may register multiple times

SEVERITY

MEDIUM

DESCRIPTION

A user can register an address multiple times since there are no restrictions in place to prevent it.

This could lead to a series of actions that might cause issues.

LOCATION OF AFFECTED CODE

```
Initiator::registerSubscription L#13
```

```
    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,
//@audit
            erc20Token: _erc20Token
        });
        subscriptionBySubscriber[_subscriber] = sub;
        subscribers.push(_subscriber);
    }
```

POC

Objective:

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

POC (ECHIDNA)

```
function test_subscription_multiple_registration() public {
    uint256 _amount = 1;
    uint256 _validUntil = 5;
    uint256 _paymentInterval = 1;
    uint256 repeat = 200;

    for (uint256 index; index < repeat; index++) {
        hevm.prank(_subscriber);
        Debugger.log("Gas Left: ", gasleft());
        initiator.registerSubscription(_subscriber, _amount, _validUntil,
_paymentInterval, _erc20Token);
    }
}
```

POC (FOUNDRY) ⇒ FUZZ

```
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");
    }
}
```

POC (HALMOS) ⇒ FUZZ

```
////////////////////////////////////////////////////////////////////////
// registerSubscription > 1
////////////////////////////////////////////////////////////////////////
function check_testFuzzMultipleSubscriptions() public {
    address subscriber = holders[0];
    uint256 iterations = 2;

    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.assume(amount > 0);
        vm.assume(validUntil > block.timestamp);
        vm.assume(paymentInterval > 0);

        vm.prank(subscriber);
```

```
        initiator.registerSubscription(subscriber, amount, validUntil,
paymentInterval, address(token));
        address[] memory registeredSubscribers =
initiator.getSubscribers();
        console.log("Subscriber i",registeredSubscribers[i]);
        console.log("==================");
        assertEq(registeredSubscribers[i], subscriber);
    }
}
```

RECOMMENDATION

Implement a requirement to check if the same user has been registered previously.

- To address the issue of users being able to register multiple times in the **Initiator** contract.
- To enhance gas efficiency alongside resolving the registration issue.
- A mapping named **subscriberStatus** has been implemented, pairing each subscriber's address with a uint256 value.
- Constants **NOT_SUBSCRIBED** and **SUBSCRIBED** are used within the mapping to denote the subscriber's registration status.
- Before a new subscription registration is processed, the contract verifies through this mapping whether the user has already registered.
- If a user is found to be **SUBSCRIBED**, the contract operation is reverted to prevent duplicate registrations.
- This approach not only resolves the problem of multiple registrations by the same user but also optimizes subscription management.
- The decision to utilize this type of mapping instead of a boolean was made to significantly reduce gas consumption.

```
mapping(address => uint256) private subscriberStatus;
```

```
// Constants to represent subscription status
uint256 private constant NOT_SUBSCRIBED = 1;
uint256 private constant SUBSCRIBED = 2;

event SubscriptionRegistered(address indexed subscriber);
event SubscriptionRemoved(address indexed subscriber);
```

```
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] != SUBSCRIBED, "Subscriber
already registered");

    ISubExecutor.SubStorage memory sub = ISubExecutor.SubStorage({
        amount: _amount,
        validUntil: _validUntil,
        validAfter: block.timestamp,
        paymentInterval: _paymentInterval,
        subscriber: _subscriber,
        initiator: address(this),
        erc20TokensValid: _erc20Token != address(0),
        erc20Token: _erc20Token
    });
    subscriptionBySubscriber[_subscriber] = sub;
    subscriberStatus[_subscriber] = SUBSCRIBED;

    emit SubscriptionRegistered(_subscriber);
}
```

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

    delete subscriptionBySubscriber[_subscriber];
    subscriberStatus[_subscriber] = NOT_SUBSCRIBED;

    emit SubscriptionRemoved(_subscriber);
}
```

```
 function initiatePayment(address _subscriber) public nonReentrant {
        require(subscriberStatus[_subscriber] == SUBSCRIBED, "Subscriber
not registered");
```

TEST

```
function test_failRegisterSubscriptionIfSubscriptionExists() public {
//@audit
    address subscriber = holders[0];
    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.expectRevert();
    vm.prank(subscriber);
    initiator.registerSubscription(subscriber, amount, validUntil,
paymentInterval, address(token));
}
```

```
Running 1 test for
test/Initiator/FoundryInitiator.t.sol:FoundryInitiatorTest
[PASS] test_failRegisterSubscriptionIfSubscriptionExists() (gas: 196535)
Traces:
  [196535]
FoundryInitiatorTest::test_failRegisterSubscriptionIfSubscriptionExists()
    ├─ [0] VM::prank(0x0000000000000000000000000000000000001001)
    │   └─ ← ()
    ├─ [179323]
Initiator::registerSubscription(0x0000000000000000000000000000000000001001
, 100000000000000000 [1e18], 2592001 [2.592e6], 864000 [8.64e5],
MockERC20: [0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
    │   ├─ emit SubscriptionRegistered(subscriber:
0x0000000000000000000000000000000000001001)
    │   └─ ← ()
    ├─ [0] VM::expectRevert(custom error f4844814:)
    │   └─ ← ()
    ├─ [0] VM::prank(0x0000000000000000000000000000000000001001)
    │   └─ ← ()
    ├─ [924]
Initiator::registerSubscription(0x0000000000000000000000000000000000001001
, 100000000000000000 [1e18], 2592001 [2.592e6], 864000 [8.64e5],
MockERC20: [0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
    │   └─ ← revert: Subscriber already registered
    └─ ← ()
```

TEAM RESPONSE

(To be filled with client's response)

2. You can register a non-ERC20 token.

SEVERITY

MEDIUM

DESCRIPTION

The function doesn't have any specific requirements when it comes to inputting the token address:

- The user could add a different token other than ERC20, such as:
  - Address of a malicious user
  - Address of a malicious contract
  - Any standard
- The only requirement it makes to differentiate between an ERC20 token and another is to compare it with:
  - `erc20TokensValid: _erc20Token == address(0) ? false : true`,
  - This simply sets it to true for any address other than address(0).
- Associated problems:
  - When calling the initiatePayment function:

```
//Check whether it's a native payment or ERC20 or ERC721
if (sub.erc20TokensValid) {
    _processERC20Payment(sub);
} else {
    _processNativePayment(sub);
}
```

- The function will never call the _processNativePayment function because the only condition for calling it is if `sub.erc20TokensValid == false`, and this condition will only be met if the `address == address(0)`.

This could lead to problems, such as:

- The `function _processNativePayment` will never be called because the only condition for it to be called is if `sub.erc20TokensValid == false`, and this condition will only be met if the `address == address(0)`.

LOCATION OF AFFECTED CODE

```
Initiator::registerSubscription L#31
```

```
erc20TokensValid: _erc20Token == address(0) ? false : true,
```

```
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,
//@audit
        erc20Token: _erc20Token
    });
    subscriptionBySubscriber[_subscriber] = sub;
    subscribers.push(_subscriber);
}
```

POC

POC (Halmos)⇒ You can register a token ≠ ERC20 ⇒ sub.erc20TokensValid == true

- **Objective**: To test if a non-ERC20 token can be registered in the contract.
- **Procedure**:
  - A token address (`FalseToken`), which is not an ERC20 token, is used to register a subscription.
  - The contract is expected to incorrectly mark `FalseToken` as a valid ERC20 token due to lack of proper validation.
- **Result**:
  - The contract mistakenly sets `erc20TokensValid` to `true` for `FalseToken`, highlighting a vulnerability.

```
////////////////////////////////////////////////////////////////////////
// registerSubscription Token != ERC20  => erc20TokensValid == true
////////////////////////////////////////////////////////////////////////
function check_test_failTokenFalse(
    uint256 amount,
    uint256 _validUntil,
    uint256 paymentInterval,
    address FalseToken) public {

    address subscriber = holders[0];

    vm.assume (1 ether <= amount && amount <= 1000 ether);
    vm.assume (1 days <= paymentInterval && paymentInterval <= 365 days);
    vm.assume (1 days <= _validUntil && _validUntil <= 365 days);
    vm.assume (FalseToken != address(token));
```

```
    uint256 validUntil = block.timestamp + _validUntil;
    vm.assume(amount > 0 && paymentInterval > 0 && validUntil >
block.timestamp);

    vm.prank(subscriber);
    bool hasFailed = false;
    try initiator.registerSubscription(subscriber, amount, validUntil,
paymentInterval, FalseToken) {

    } catch {
        hasFailed = true
    }

    if (hasFailed) {
        fail("Revert");
    }

    ISubExecutor.SubStorage memory sub =
initiator.getSubscription(subscriber);
    assertEq(sub.amount, amount);
    assertEq(sub.validUntil, validUntil);
    assertEq(sub.paymentInterval, paymentInterval);
    assertEq(sub.subscriber, subscriber);
    assertEq(sub.initiator, address(initiator));
    assertEq(sub.erc20Token, address(FalseToken));
    assertEq(sub.erc20TokensValid, FalseToken != address(0));
}
```

POC (Foundry) Fuzz⇒ You can register a token ≠ ERC20 ⇒ sub.erc20TokensValid == true

- **Objective**: To use 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);

}
```

POC (Halmos)You cannot register a token ≠ ⇒ sub.erc20TokensValid == false

- **Objective**: To test the behavior of registering a subscription using the zero address (`address(0)`).
- **Procedure**:
  - A subscription is registered with `address0` (zero address) as the token.
  - The contract should correctly set `erc20TokensValid` to `false`.
- **Result**:
  - The contract recognizes `address0` as not an ERC20 token, thereby setting `erc20TokensValid` to `false`.

```
////////////////////////////////////////////////////////////////////////
// registerSubscription NoToken ERC20 => address(0) => erc20TokensValid ==
false
////////////////////////////////////////////////////////////////////////
function check_test_Address0_failTokenFalse(
    uint256 amount,
    uint256 _validUntil,
    uint256 paymentInterval,
    address address0) public {

    address subscriber = holders[0];

    vm.assume (1 ether <= amount && amount <= 1000 ether);
    vm.assume (1 days <= paymentInterval && paymentInterval <= 365 days);
    vm.assume (1 days <= _validUntil && _validUntil <= 365 days);
    vm.assume (address0 == address(0));

    uint256 validUntil = block.timestamp + _validUntil;
    vm.assume(amount > 0 && paymentInterval > 0 && validUntil >
block.timestamp);

    vm.prank(subscriber);
    bool hasFailed = false;
    try initiator.registerSubscription(subscriber, amount, validUntil,
paymentInterval, address0) {
    } catch {
        hasFailed = true;
```

```
        }

        if (hasFailed) {
            fail("Revert");
        }

        ISubExecutor.SubStorage memory sub =
    initiator.getSubscription(subscriber);
        assertEq(sub.amount, amount);
        assertEq(sub.validUntil, validUntil);
        assertEq(sub.paymentInterval, paymentInterval);
        assertEq(sub.subscriber, subscriber);
        assertEq(sub.initiator, address(initiator));
        assertEq(sub.erc20Token, address(address0));
        assertEq(sub.erc20TokensValid, false);
    }
```

RECOMMENDATION

The solution is to create a requirement that clearly distinguishes between valid tokens, other types of tokens, and invalid tokens.

This way, users will only be able to register valid tokens on the platform, and they won't be able to register malicious tokens, any address, or any standards different from the accepted ones.

It should implement logic to differentiate the type of token registered on the platform, as with the current logic, any standard type of token can be registered. It only restricts if it is an address(0), and even then, it allows registration without issues.

TEAM RESPONSE

(To be filled with client's response)

3. The function _processNativePayment will never be called.

SEVERITY

MEDIUM

DESCRIPTION

As mentioned in the previous error, this leads to payments being made only through the _processERC20Payment function, as the only condition for calling this function is that the address is not address(0).

In the case of having a different standard that does not implement ERC20, it would still call the same function instead of the one it should, as the only condition for calling the _processNativePayment function is that it is an address(0).

- The function `_processNativePayment` will never be called because the only condition for it to be called is if `sub.erc20TokensValid == false`, and this condition will only be met if the `address`

`== address(0)`.

- Therefore, when performing an `initiatePayment` ⇒ `processPayment`:
  - It will always call `_processERC20Payment` because it is different from address(0).
  - And it will never call `_processNativePayment` since it will never be an address(0) by default.

LOCATION OF AFFECTED CODE

```
Initiator::initiatePayment L#53
```

```
//Check whether it's a native payment or ERC20 or ERC721
if (sub.erc20TokensValid) {
    _processERC20Payment(sub);
} else {
    _processNativePayment(sub);
}
```

```
function initiatePayment(address _subscriber) public nonReentrant {
    ISubExecutor.SubStorage storage subscription =
subscriptionBySubscriber[_subscriber];

    require(subscription.validUntil > block.timestamp, "Subscription is
not active");
    require(subscription.validAfter < block.timestamp, "Subscription is
not active");
    require(subscription.amount > 0, "Subscription amount is 0");
    require(subscription.paymentInterval > 0, "Payment interval is 0");

    // uint256 lastPaid =
ISubExecutor(subscription.subscriber).getLastPaidTimestamp(address(this));
    // if (lastPaid != 0) {
    //     require(lastPaid + subscription.paymentInterval >
block.timestamp, "Payment interval not yet reached");
    // }
    ISubExecutor(subscription.subscriber).processPayment();
}
```

```
function processPayment() external nonReentrant {
    SubStorage storage sub = getKernelStorage().subscriptions[msg.sender];
    require(block.timestamp >= sub.validAfter, "Subscription not yet
valid");
    require(block.timestamp <= sub.validUntil, "Subscription expired");
    require(msg.sender == sub.initiator, "Only the initiator can initiate
payments");

    //Check when the last payment was done
```

```
    PaymentRecord[] storage paymentHistory =
getKernelStorage().paymentRecords[msg.sender];
    if (paymentHistory.length > 0) {
        PaymentRecord storage lastPayment =
paymentHistory[paymentHistory.length - 1];
        require(block.timestamp >= lastPayment.timestamp +
sub.paymentInterval, "Payment interval not yet reached");
    } else {
        require(block.timestamp >= sub.validAfter + sub.paymentInterval,
"Payment interval not yet reached");
    }


getKernelStorage().paymentRecords[msg.sender].push(PaymentRecord(sub.amoun
t, block.timestamp, sub.subscriber));

    //Check whether it's a native payment or ERC20 or ERC721
    if (sub.erc20TokensValid) {
        _processERC20Payment(sub);
    } else {
        _processNativePayment(sub);
    }

    emit paymentProcessed(msg.sender, sub.amount);
}
```

POC

POC Foundry Fuzz

```
function test_Fuzz_Address0_InitiatePayment(
    uint256 amount,
    uint256 validUntilOffsetDays,
    uint256 paymentIntervalDays,
    uint256 tokenAmount,
    address FalseToken
) public {
    // Ensure reasonable input values.
    vm.assume(amount > 0 && amount <= 100 ether);
    vm.assume(validUntilOffsetDays >= 1 && validUntilOffsetDays <= 365);
    vm.assume(paymentIntervalDays >= 1 && paymentIntervalDays <=
validUntilOffsetDays);
    vm.assume(tokenAmount > amount && tokenAmount <= 1000 ether);
    FalseToken = address(0);

    address subscriber = deployer;
    uint256 _validAfter = block.timestamp + 1 days;
    uint256 validUntil = _validAfter + validUntilOffsetDays * 1 days;
    uint256 paymentInterval = paymentIntervalDays * 1 days;

    // Transfer tokens to the Initiator contract as part of the test
```

```
setup.
    uint256 tokenBalance = amount + tokenAmount;
    vm.startPrank(deployer);
    token.mint(address(subExecutor), tokenBalance);
    vm.stopPrank();

    // Approve Initiator to spend tokens on behalf of SubExecutor.
    vm.startPrank(address(subExecutor));
    token.approve(address(initiator), tokenBalance);
    vm.stopPrank();

    // Register the subscription with fuzzing parameters.
    vm.prank(subscriber);
    initiator.registerSubscription(subscriber, amount, validUntil,
paymentInterval, address(FalseToken));
    initiator.setSubExecutor(address(subExecutor));

    // Advance to the time when the subscription is active but not
expired.
    uint256 warpToTime = _validAfter + (validUntilOffsetDays / 2) * 1
days; // Halfway to ensure it's active.
    vm.warp(warpToTime);

    // Attempt to initiate a payment as the subscriber.
    vm.prank(subscriber);
    bool success;
    try initiator.initiatePayment(subscriber) {
        success = true;
    } catch {
        success = false;
    }

    assertTrue(success, "The call to initiatePayment should have been
successful");
}
```

RECOMMENDATION

Create a requirement that clearly distinguishes between valid tokens, other types of tokens, and invalid tokens.

This way, users will only be able to register valid tokens on the platform, and they won't be able to register malicious tokens, any address, or any standards different from the accepted ones.

TEAM RESPONSE

(To be filled with client's response)

4. Potential payment execution failures due to incorrect call to `processPayment()`

SEVERITY

HIGH

DESCRIPTION

The identified vulnerability focuses on the incorrect implementation of the ISubExecutor interface, specifically in the context of the processPayment functionality.

The issue lies in incorrectly assuming that the subscriber's address is a contract that implements ISubExecutor. This assumption may not hold true, especially if the subscriber is an externally owned account (EOA) or a contract that does not adhere to this interface.

This fundamental misunderstanding can lead to failures in payment execution and directly impact the subscription and payment logic of the system.

- **Inadequate Interface Definition**: The `ISubExecutor` interface must be correctly defined to include all necessary functions, such as `processPayment`, and any additional functions that may be required.
- **Incorrect Implementation in Subscriber Contracts**: It is wrongly assumed that the subscriber's address is a contract that implements `ISubExecutor`, which may not always be the case, especially if the subscriber is an EOA or a contract that does not follow this interface.
- **Incorrect Calls to `processPayment`**: The intention to call the `processPayment` method on a specific contract is not adequately fulfilled by attempting to make this call on the subscriber's address.

Impact

- **Payment Execution Failure:** Inability to correctly call processPayment prevents payments from being executed as intended, which may result in service disruption or failure to compensate expected subscribers or initiators.
- **Security and Access Control:** Attempting to call processPayment on addresses that do not implement ISubExecutor (or even on EOAs) could pose a security risk or, at the very least, result in unnecessary gas consumption without achieving the desired effect.
- **Design Rigidity:** The presumption that all subscribers must be contracts implementing ISubExecutor reduces the system's flexibility and its ability to adapt to different subscription models.

LOCATION OF AFFECTED CODE

Initiator::initiatePayment L#65

```
 ISubExecutor(subscription.subscriber).processPayment();
```

```
function initiatePayment(address _subscriber) public nonReentrant {
    ISubExecutor.SubStorage storage subscription =
subscriptionBySubscriber[_subscriber];
    require(subscription.validUntil > block.timestamp, "Subscription is
not active");
    require(subscription.validAfter < block.timestamp, "Subscription is
```

```
not active");
    require(subscription.amount > 0, "Subscription amount is 0");
    require(subscription.paymentInterval > 0, "Payment interval is 0");

    // uint256 lastPaid =
ISubExecutor(subscription.subscriber).getLastPaidTimestamp(address(this));
    // if (lastPaid != 0) {
    //     require(lastPaid + subscription.paymentInterval >
block.timestamp, "Payment interval not yet reached");
    // }

    ISubExecutor(subscription.subscriber).processPayment();
}
```

POC

We create these POCs to demonstrate that even when all the requirements of the function are correct at the time of calling the interface, it reverts due to the poor implementation.

That's why we have added console.log statements within the function to visualize where it reverts.

```
function initiatePayment(address _subscriber) public nonReentrant {
    ISubExecutor.SubStorage storage subscription =
subscriptionBySubscriber[_subscriber];

    console.log("============" );
    console.log("validUntil ",subscription.validUntil);
    console.log("> block.timestamp",block.timestamp );
    console.log("============" );

    console.log("validAfter ",subscription.validAfter );
    console.log("< block.timestamp",block.timestamp );
    console.log("============" );

    console.log("amount > 0? =>",subscription.amount );
    console.log("============" );

    console.log("paymentInterval > 0? =>",subscription.paymentInterval );
    console.log("============" );
    console.log("////////////////////////////////////////" );

    require(subscription.validUntil > block.timestamp, "Subscription is
not active");
    require(subscription.validAfter < block.timestamp, "Subscription is
not active");
    require(subscription.amount > 0, "Subscription amount is 0");
    require(subscription.paymentInterval > 0, "Payment interval is 0");

    console.log("THE CALL HAS PASSED ALL REQUIREMENTS" );
```

```
    // uint256 lastPaid =
ISubExecutor(subscription.subscriber).getLastPaidTimestamp(address(this));
    // if (lastPaid != 0) {
    //     require(lastPaid + subscription.paymentInterval >
block.timestamp, "Payment interval not yet reached");
    // }

    ISubExecutor(subscription.subscriber).processPayment();
}
```

Foundry UNIT

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


    uint256 tokenAmount = 10 ether;
    vm.prank(subscriber);
    token.transfer(address(initiator), tokenAmount);

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

    uint256 warpToTime = _validAfter + 10;
    vm.warp(warpToTime);

    vm.prank(subscriber);
    bool success;
    try initiator.initiatePayment(subscriber) {
        success = true;
    } catch {
        success = false;
    }

    assertTrue(success, "InitiatePayment call successful");
}
```

RESULT

```
[FAIL. Reason: assertion failed] test_initiatePayment_ActiveSubscription()
(gas: 280392)

  Error: InitiatePayment call successful
```

```
  Error: Assertion Failed

Traces:
  [280392] FoundryInitiatorTest::test_initiatePayment_ActiveSubscription()
    ├─ [0] VM::prank(deployer:
[0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946])
    │   └─ ← ()
    ├─ [29810] MockERC20::transfer(Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], 10000000000000000000 [1e19])
    │   ├─ emit Transfer(from: deployer:
[0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946], to: Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], value: 10000000000000000000
[1e19])
    │   └─ ← true
    ├─ [0] VM::prank(deployer:
[0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946])
    │   └─ ← ()
    ├─ [200320] Initiator::registerSubscription(deployer:
[0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946], 100000000000000000 [1e18],
950401 [9.504e5], 864000 [8.64e5], MockERC20:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
    │   └─ ← ()
    ├─ [1932] Initiator::getSubscription(deployer:
[0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946]) [staticcall]
    │   └─ ← SubStorage({ amount: 100000000000000000 [1e18], validUntil:
950401 [9.504e5], validAfter: 1, paymentInterval: 864000 [8.64e5],
subscriber: 0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946, initiator:
0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b, erc20TokensValid: true,
erc20Token: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264 })
    ├─ [0] VM::warp(86411 [8.641e4])
    │   └─ ← ()
    ├─ [0] VM::prank(deployer:
[0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946])
    │   └─ ← ()
    ├─ [18341] Initiator::initiatePayment(deployer:
[0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946])

    │   └─ ← EvmError: Revert
    ├─ emit log_named_string(key: "Error", val: "InitiatePayment call
successful.")
    ├─ emit log(val: "Error: Assertion Failed")
    ├─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
0x6661696c6564000000000000000000000000000000000000000000000000000000,
0x0000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← ()
    └─ ← ()
```

```
Logs:
  ============
  validUntil  950401
  > block.timestamp 86411
```

```
  ============
  validAfter  1
  < block.timestamp 86411
  ============
  amount > 0? => 1000000000000000000
  ============
  paymentInterval > 0? => 864000
  ============
  //////////////////////////////////////
  THE CALL HAS PASSED ALL REQUIREMENTS
```

FOUNDRY FUZZ

```
function test_Fuzz_tinitiatePayment_ActiveSubscription(
    uint256 amount,
    uint256 _validUntil,
    uint256 paymentInterval,
    address FalseToken) public {


    address subscriber = holders[0];
    uint256 _validAfter = block.timestamp + 1 days;
    uint256 validUntil = _validAfter + 10 days;

    amount = bound(amount, 1 ether, 1000 ether);
    paymentInterval = bound(paymentInterval, 1 days, 365 days);

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

    ISubExecutor.SubStorage memory sub =
initiator.getSubscription(subscriber);
    assertEq(sub.amount, amount);
    assertEq(sub.validUntil, validUntil);
    assertEq(sub.paymentInterval, paymentInterval);
    assertEq(sub.subscriber, subscriber);
    assertEq(sub.initiator, address(initiator));
    assertEq(sub.erc20Token, address(FalseToken));
    assertEq(sub.erc20TokensValid, FalseToken != address(0));

    uint256 warpToTime = _validAfter + 10;
    vm.warp(warpToTime);

    vm.prank(subscriber);
    bool success;
    try initiator.initiatePayment(subscriber) {
        success = true;
    } catch {
        success = false;
    }
```

```
    assertTrue(success, "InitiatePayment call successful");
}
```

RESULT

```
[FAIL. Reason: assertion failed; counterexample:
calldata=0xad8465950000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000 args=[0, 0, 0,
0x0000000000000000000000000000000000000000]]
test_Fuzz_tinitiatePayment_ActiveSubscription(uint256,uint256,uint256,addr
ess) (runs: 0, μ: 0, ~: 0)

  Error: InitiatePayment call successful
  Error: Assertion Failed

Traces:
  [295186]
FoundryInitiatorTest::test_Fuzz_tinitiatePayment_ActiveSubscription(0, 0,
0, 0x0000000000000000000000000000000000000000)
    ├─ [0] VM::prank(0x0000000000000000000000000000000000001001)
    │   └─ ← ()
    ├─ [200320]
Initiator::registerSubscription(0x0000000000000000000000000000000000001001
, 100000000000000000 [1e18], 950401 [9.504e5], 86400 [8.64e4], MockERC20:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
    │   └─ ← ()
    ├─ [1932]
Initiator::getSubscription(0x0000000000000000000000000000000000001001)
[staticcall]
    │   └─ ← SubStorage({ amount: 100000000000000000 [1e18], validUntil:
950401 [9.504e5], validAfter: 1, paymentInterval: 86400 [8.64e4],
subscriber: 0x0000000000000000000000000000000000001001, initiator:
0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b, erc20TokensValid: true,
erc20Token: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264 })
    ├─ emit log(val: "Error: a == b not satisfied [address]")
    ├─ emit log_named_address(key: "      Left", val: MockERC20:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
    ├─ emit log_named_address(key: "     Right", val:
0x0000000000000000000000000000000000000000)
    ├─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
0x6661696c6564000000000000000000000000000000000000000000000000000000,
0x0000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← ()
    ├─ emit log(val: "Error: a == b not satisfied [bool]")
    ├─ emit log_named_string(key: "      Left", val: "true")
    ├─ emit log_named_string(key: "     Right", val: "false")
    ├─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
0x6661696c6564000000000000000000000000000000000000000000000000000000,
```

```
0x0000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← ()
    ├─ [0] VM::warp(86411 [8.641e4])
    │   └─ ← ()
    ├─ [0] VM::prank(0x0000000000000000000000000000000000001001)
    │   └─ ← ()
    ├─ [23199] Initiator::setSubExecutor(SubExecutor:
[0xfF2Bd636B9Fc89645C2D336aeaDE2E4AbaFe1eA5])
    │   ├─ [0] console::log("He pasado") [staticcall]
    │   │   └─ ← ()
    │   └─ ← ()
    ├─ [15841]
Initiator::initiatePayment(0x0000000000000000000000000000000000001001)

    │   └─ ← EvmError: Revert
    ├─ emit log_named_string(key: "Error", val: "InitiatePayment call
successful")
    ├─ emit log(val: "Error: Assertion Failed")
    ├─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
0x6661696c6564000000000000000000000000000000000000000000000000000,
0x0000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← ()
    └─ ← ()
```

```
Logs:
  Bound Result 100000000000000000
  Bound Result 86400
  Error: a == b not satisfied [address]
        Left: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264
       Right: 0x0000000000000000000000000000000000000000
  Error: a == b not satisfied [bool]
        Left: true
       Right: false
  ============
  validUntil  950401
  > block.timestamp 86411
  ============
  validAfter  1
  < block.timestamp 86411
  ============
  amount > 0? => 100000000000000000
  ============
  paymentInterval > 0? => 86400
  ============
  ////////////////////////////////////////
  THE CALL HAS PASSED ALL REQUIREMENTS
```

RECOMMENDATION

1. **Storing Reference to the `SubExecutor` Contract**: It is suggested to store a reference to the `SubExecutor` contract within the `Initiator` contract to facilitate direct calls to `processPayment`.
2. **Method Call Adjustments**: The call to `processPayment` should be adjusted to use the stored reference to the `SubExecutor` contract, instead of attempting to make the call to the subscriber's address.
3. **Adequate Initialization and Configuration**: It is crucial to ensure that all configurations and the subscription initialization are correctly done to allow proper functioning of `processPayment`.
4. **Security and Validations**: The importance of performing adequate validations before calls to `processPayment` and considering security implications when interacting with external contracts is emphasized.

```
function setSubExecutor(address _subExecutorAddress) external  {
    require(_subExecutorAddress != address(0));
    subExecutor = ISubExecutor(_subExecutorAddress);
}
```

```
function initiatePayment(address _subscriber) public nonReentrant {
    ISubExecutor.SubStorage storage subscription =
subscriptionBySubscriber[_subscriber];

    require(subscription.validUntil > block.timestamp, "Subscription is
not active");
    require(subscription.validAfter < block.timestamp, "Subscription is
not active");
    require(subscription.amount > 0, "Subscription amount is 0");
    require(subscription.paymentInterval > 0, "Payment interval is 0");


    // uint256 lastPaid =
ISubExecutor(subscription.subscriber).getLastPaidTimestamp(address(this));
    // if (lastPaid != 0) {
    //     require(lastPaid + subscription.paymentInterval >
block.timestamp, "Payment interval not yet reached");
    // }

    subExecutor.processPayment();
}
```

RESULT

```
Traces:
 [320104]
FoundryInitiatorTest::test_Fuzz_tinitiatePayment_ActiveSubscription(0, 0,
0, 0x0000000000000000000000000000000000000000)
   ├─ [0] console::log("Bound Result", 1000000000000000000 [1e18])
```

```
  [staticcall]
    │   └─ ← ()
    ├─ [0] console::log("Bound Result", 86400 [8.64e4]) [staticcall]
    │   └─ ← ()
    ├─ [0] VM::prank(0x0000000000000000000000000000000000001001)
    │   └─ ← ()
    ├─ [200320]
  Initiator::registerSubscription(0x0000000000000000000000000000000000001001
  , 100000000000000000 [1e18], 950401 [9.504e5], 86400 [8.64e4], MockERC20:
  [0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
    │   └─ ← ()
    ├─ [1932]
  Initiator::getSubscription(0x0000000000000000000000000000000000001001)
  [staticcall]
    │   └─ ← SubStorage({ amount: 100000000000000000 [1e18], validUntil:
  950401 [9.504e5], validAfter: 1, paymentInterval: 86400 [8.64e4],
  subscriber: 0x0000000000000000000000000000000000001001, initiator:
  0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b, erc20TokensValid: true,
  erc20Token: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264 })
    ├─ emit log(val: "Error: a == b not satisfied [address]")
    ├─ emit log_named_address(key: "      Left", val: MockERC20:
  [0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
    ├─ emit log_named_address(key: "     Right", val:
  0x0000000000000000000000000000000000000000)
    ├─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
  0x6661696c65640000000000000000000000000000000000000000000000000000,
  0x0000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← ()
    ├─ emit log(val: "Error: a == b not satisfied [bool]")
    ├─ emit log_named_string(key: "      Left", val: "true")
    ├─ emit log_named_string(key: "     Right", val: "false")
    ├─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
  0x6661696c65640000000000000000000000000000000000000000000000000000,
  0x0000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← ()
    ├─ [0] VM::warp(86411 [8.641e4])
    │   └─ ← ()
    ├─ [0] VM::prank(0x0000000000000000000000000000000000001001)
    │   └─ ← ()
    ├─ [23199] Initiator::setSubExecutor(SubExecutor:
  [0xfF2Bd636B9Fc89645C2D336aeaDE2E4AbaFe1eA5])
    │   ├─ [0] console::log("OK setSubExecutor") [staticcall]
    │   │   └─ ← ()
    │   └─ ← ()
    ├─ [41092]
  Initiator::initiatePayment(0x0000000000000000000000000000000000001001)
    │   ├─ [0] console::log("THE CALL HAS PASSED ALL REQUIREMENTS")
  [staticcall]
    │   │   └─ ← ()
    │   ├─ [22608] SubExecutor::processPayment()
    │   │   ├─ [0] console::log("Within processPayment") [staticcall]
    │   │   │   └─ ← ()
    │   │   └─ ← revert: Subscription expired
    │   └─ ← revert: Subscription expired
```

```
    ├─ emit log_named_string(key: "Error", val: "InitiatePayment call
successful")
    ├─ emit log(val: "Error: Assertion Failed")
    ├─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
0x6661696c65640000000000000000000000000000000000000000000000000000,
0x0000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← ()
    └─ ← ()
```

TEAM RESPONSE

(To be filled with client's response)

5. Inability to read any registered subscription’s data blocks payment

SEVERITY

HIGH

DESCRIPTION

An issue in the implementation is not allowing to read in `initiatePayment()` any of the stored values from `registerSubscription()`.

Contract Storage Logic Issue

The storage logic within the contract is flawed due to the way the ISubExecutor interface is implemented. This issue arises because the values stored in functions are incorrectly initialized within the struct of the interface.

IMPACT

- **Registration Function**: For example, when registering a user using the `registerSubscription` function, the values are stored within the interface, specifically in the `ISubExecutor.SubStorage` struct. At this point, the values seem to be properly registered within the struct of the interface.
- **InitiatePayment Function**: However, problems occur when attempting to call the `initiatePayment` function. This function involves calling the SubExecutor contract (which was previously fixed from a prior error). After ensuring that all the function requirements are met and verifying correctness, the function proceeds to call the `processPayment` function of the SubExecutor contract.

Contract Responsibilities

- **Initiator Contract**: This contract is responsible for storing data in a struct of the SubExecutor interface.
- **SubExecutor Contract**: In contrast, the SubExecutor contract is responsible for storing data in the KernelStorage contract.

LOCATION OF AFFECTED CODE

`Initiator::initiatePayment`

POC

Upon debugging, it becomes apparent that all the user data is empty when the `require` statements are executed.

FOUNDRY FUZZ

```
function test_Fuzz_initiatePayment_ActiveSubscription(
    uint256 amount,
    uint256 _validUntil,
    uint256 paymentInterval,
    address FalseToken) public {


    address subscriber = holders[0];
    uint256 _validAfter = block.timestamp + 1 days;
    uint256 validUntil = _validAfter + 10 days;

    amount = bound(amount, 1 ether, 1000 ether);
    paymentInterval = bound(paymentInterval, 1 days, 365 days);

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

    ISubExecutor.SubStorage memory sub =
initiator.getSubscription(subscriber);
    assertEq(sub.amount, amount);
    assertEq(sub.validUntil, validUntil);
    assertEq(sub.paymentInterval, paymentInterval);
    assertEq(sub.subscriber, subscriber);
    assertEq(sub.initiator, address(initiator));
    assertEq(sub.erc20Token, address(FalseToken));
    assertEq(sub.erc20TokensValid, FalseToken != address(0));

    uint256 warpToTime = _validAfter + 10;
    vm.warp(warpToTime);

    vm.prank(subscriber);
    initiator.setSubExecutor(address(subExecutor));
    bool success;
    try initiator.initiatePayment(subscriber) {
        success = true;
    } catch {
        success = false;
    }

    assertTrue(success, "InitiatePayment call successful");
}
```

RESULT

```
[FAIL. Reason: assertion failed; counterexample:
calldata=0xad8465950000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000 args=[0, 0, 0,
0x0000000000000000000000000000000000000000]]
test_Fuzz_tinitiatePayment_ActiveSubscription(uint256,uint256,uint256,addr
ess) (runs: 0, μ: 0, ~: 0)
  Error: a == b not satisfied [address]
        Left: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264
       Right: 0x0000000000000000000000000000000000000000
  Error: a == b not satisfied [bool]
        Left: true
       Right: false
  Error: InitiatePayment call successful
  Error: Assertion Failed

Traces:
  [340537]
FoundryInitiatorTest::test_Fuzz_tinitiatePayment_ActiveSubscription(0, 0,
0, 0x0000000000000000000000000000000000000000)
    ├─ [0] VM::prank(0x0000000000000000000000000000000000001001)
    │   └─ ← ()
    ├─ [200320]
Initiator::registerSubscription(0x0000000000000000000000000000000000001001
, 100000000000000000 [1e18], 950401 [9.504e5], 86400 [8.64e4], MockERC20:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
    │   └─ ← ()
    ├─ [1932]
Initiator::getSubscription(0x0000000000000000000000000000000000001001)
[staticcall]
    │   └─ ← SubStorage({ amount: 100000000000000000 [1e18], validUntil:
950401 [9.504e5], validAfter: 1, paymentInterval: 86400 [8.64e4],
subscriber: 0x0000000000000000000000000000000000001001, initiator:
0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b, erc20TokensValid: true,
erc20Token: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264 })
    ├─ emit log(val: "Error: a == b not satisfied [address]")
    ├─ emit log_named_address(key: "      Left", val: MockERC20:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
    ├─ emit log_named_address(key: "     Right", val:
0x0000000000000000000000000000000000000000)
    ├─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
0x6661696c6564000000000000000000000000000000000000000000000000000000,
0x0000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← ()
    ├─ emit log(val: "Error: a == b not satisfied [bool]")
    ├─ emit log_named_string(key: "      Left", val: "true")
    ├─ emit log_named_string(key: "     Right", val: "false")
    ├─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
0x6661696c6564000000000000000000000000000000000000000000000000000000,
```

```
0x0000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← ()
    ├─ [1932]
Initiator::getSubscription(0x0000000000000000000000000000000000001001)
[staticcall]
    │   └─ ← SubStorage({ amount: 100000000000000000 [1e18], validUntil:
950401 [9.504e5], validAfter: 1, paymentInterval: 86400 [8.64e4],
subscriber: 0x0000000000000000000000000000000000001001, initiator:
0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b, erc20TokensValid: true,
erc20Token: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264 })
    ├─ [23199] Initiator::setSubExecutor(SubExecutor:
[0xfF2Bd636B9Fc89645C2D336aeaDE2E4AbaFe1eA5])
    │   ├─ [0] console::log("OK setSubExecutor") [staticcall]
    │   │   └─ ← ()
    │   └─ ← ()
    ├─ [16023]
SubExecutor::getSubscription(0x0000000000000000000000000000000000001001)
[staticcall]
    │   └─ ← SubStorage({ amount: 0, validUntil: 0, validAfter: 0,
paymentInterval: 0, subscriber:
0x0000000000000000000000000000000000000000, initiator:
0x0000000000000000000000000000000000000000, erc20TokensValid: false,
erc20Token: 0x0000000000000000000000000000000000000000 })
    ├─ [0] VM::warp(86411 [8.641e4])
    │   └─ ← ()
    ├─ [0] VM::prank(0x0000000000000000000000000000000000001001)
    │   └─ ← ()
    ├─ [38592]
Initiator::initiatePayment(0x0000000000000000000000000000000000001001)

    │   ├─ [22608] SubExecutor::processPayment()
    │   │   └─ ← revert: Subscription expired
    │   └─ ← revert: Subscription expired
    ├─ emit log_named_string(key: "Error", val: "InitiatePayment call
successful")
    ├─ emit log(val: "Error: Assertion Failed")
    ├─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
0x6661696c6564000000000000000000000000000000000000000000000000000000,
0x0000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← ()
    └─ ← ()
```

```
Logs:

BEFORE:
  ============
  validUntil  950401
  > block.timestamp 86411
  ============
  validAfter  1
```

```
  < block.timestamp 86411
  ============
  amount > 0? => 100000000000000000
  ============
  paymentInterval > 0? => 86400
  ============

////////////////////////////////////////
AFTER:

Within processPayment
  ============
  block.timestamp  86411
  >= validAfter  0
  ============
  block.timestamp  86411
  <= validUntil  0
  ============
  msg.sender == 0x0000000000000000000000000000000000000000
  ============
  ////////////////////////////////////////
```

EXPLANATION

Please notice in the `AFTER` section how the values of `amount`, `paymentInterval`, `validAfter` and `validUntil`, when read, are `0`.

```
1932]
Initiator::getSubscription(0x0000000000000000000000000000000000001001)
[staticcall]
    │   └─ ← SubStorage({ amount: 100000000000000000 [1e18], validUntil:
950401 [9.504e5], validAfter: 1, paymentInterval: 86400 [8.64e4],
subscriber: 0x0000000000000000000000000000000000001001, initiator:
0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b, erc20TokensValid: true,
erc20Token: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264 })
```

```
[16023]
SubExecutor::getSubscription(0x0000000000000000000000000000000000001001)
[staticcall]
    │   └─ ← SubStorage({ amount: 0, validUntil: 0, validAfter: 0,
paymentInterval: 0, subscriber:
0x0000000000000000000000000000000000000000, initiator:
0x0000000000000000000000000000000000000000, erc20TokensValid: false,
erc20Token: 0x0000000000000000000000000000000000000000 })
```

RECOMMENDATION

There is the need to refactor the interface and the write/read subscription logic entirely in order to have the expected behaviour.

TEAM RESPONSE

(To be filled with client's response)

6. _processERC20Payment Logic Flaw: Incorrect Token Transfer

SEVERITY

HIGH

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 potential locking of tokens.

The **SubExecutor** contract is designed to manage subscriptions and process automatic payments between accounts, using ERC20 tokens as the payment medium. 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** (the initiator of the subscription) using the **transferFrom** method.

Analysis of Call Flow and Identified Issues

1. Subscription Creation (createSubscription):

- The contract allows setting up a subscription, assigning an **initiator**, an amount, and a specific ERC20 token for future payments. This process is appropriate and does not present issues in itself.

2. 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. However, the method incorrectly assumes that **transferFrom** is the suitable mechanism for transferring tokens.

3. ERC20 Token Transfer (_processERC20Payment):

- The use of **transferFrom** implies that the initiator has given permission to 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

`SubExecutor::_processERC20Payment`

POC

```
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(); // Assume processPayment does not
require arguments.

    // 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.");
```

```
}
```

RESULT

```
[FAIL. Reason: revert: ERC20: insufficient allowance; counterexample:
calldata=0xe1cf30e10000000000000000000000000000000000000000000003232b83
637bcb2b900000000000000000000000000000000000000000000000000000000004ee
args=[3617155993734787769 [3.617e18], 1262]]
test_FuzzERC20Payment(uint256,uint256) (runs: 0, μ: 0, ~: 0)

Traces:
  [549843]
SFoundrySubExecutorTest::test_FuzzERC20Payment(3617155993734787769
[3.617e18], 1262)
    ├─ [0] VM::assume(true) [staticcall]
    │   └─ ← ()
    ├─ [0] VM::assume(true) [staticcall]
    │   └─ ← ()
    ├─ [0] VM::startPrank(deployer:
[0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946])
    │   └─ ← ()
    ├─ [29531] MockERC20::mint(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264], 3617155993734787769
[3.617e18])
    │   ├─ emit Transfer(from: 0x0000000000000000000000000000000000000000,
to: SubExecutor: [0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264], value:
3617155993734787769 [3.617e18])
    │   └─ ← ()
    ├─ [0] VM::stopPrank()
    │   └─ ← ()
    ├─ [0] VM::startPrank(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
    │   └─ ← ()
    ├─ [24658] MockERC20::approve(Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], 3617155993734787769
[3.617e18])
    │   ├─ emit Approval(owner: SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264], spender: Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], value: 3617155993734787769
[3.617e18])
    │   └─ ← true
    ├─ [0] VM::stopPrank()
    │   └─ ← ()
    ├─ [582] MockERC20::balanceOf(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264]) [staticcall]
    │   └─ ← 3617155993734787769 [3.617e18]
    ├─ [2582] MockERC20::balanceOf(Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b]) [staticcall]
    │   └─ ← 0
```

```
    ├─ [2556] SubExecutor::getOwner() [staticcall]
    │   └─ ← 0x0000000000000000000000000000000000000000
    ├─ [0] VM::startPrank(0x0000000000000000000000000000000000000000)
    │   └─ ← ()
    ├─ [361573] SubExecutor::createSubscription(Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], 1262, 2592000 [2.592e6],
7776001 [7.776e6], MockERC20:
[0xfF2Bd636B9Fc89645C2D336aeaDE2E4AbaFe1eA5])
    │   ├─ [200320] Initiator::registerSubscription(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264], 1262, 7776001 [7.776e6],
2592000 [2.592e6], MockERC20:
[0xfF2Bd636B9Fc89645C2D336aeaDE2E4AbaFe1eA5])
    │   │   └─ ← ()
    │   ├─ emit subscriptionCreated(_initiator:
0x0000000000000000000000000000000000000000, _subscriber: Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], _amount: 1262)
    │   └─ ← ()
    ├─ [0] VM::stopPrank()
    │   └─ ← ()
    ├─ [0] VM::warp(2592002 [2.592e6])
    │   └─ ← ()
    ├─ [0] VM::prank(Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b])
    │   └─ ← ()
    ├─ [106050] SubExecutor::processPayment()
    │   ├─ [582] MockERC20::balanceOf(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264]) [staticcall]
    │   │   └─ ← 3617155993734787769 [3.617e18]
    │   ├─ [0] console::log("ERC20 Token balance of SubExecutor:",
3617155993734787769 [3.617e18]) [staticcall]
    │   │   └─ ← ()
    │   ├─ [870] MockERC20::allowance(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264], Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b]) [staticcall]
    │   │   └─ ← 3617155993734787769 [3.617e18]
    │   ├─ [0] console::log("Allowance for Initiator to spend
SubExecutor's tokens:", 3617155993734787769 [3.617e18]) [staticcall]
    │   │   └─ ← ()
    │   ├─ [3038] MockERC20::transferFrom(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264], Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], 1262)
    │   │   └─ ← revert: ERC20: insufficient allowance
    │   └─ ← revert: ERC20: insufficient allowance
    └─ ← revert: ERC20: insufficient allowance
```

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

RECOMMENDATION

Modify Transfer Logic in _processERC20Payment:

- The solution lies in changing the **transfer** method from **transferFrom** to transfer, allowing the **SubExecutor** to send tokens directly to the **initiator** without requiring prior authorization. This change eliminates the need for complex approval and ensures that the flow of funds is direct and secure.
- Implementing this modification will simplify the management of subscriptions and payments, aligning it with standard ERC20 transfer practices and eliminating the risk of fund lockup due to authorization errors.

```
function _processERC20Payment(SubStorage storage sub) internal {
    IERC20 token = IERC20(sub.erc20Token);
    uint256 balance = token.balanceOf(address(this));
    require(balance >= sub.amount, "Insufficient token balance");
    // Key change: Use of transfer instead of transferFrom
    (bool success) = token.transfer(sub.initiator, sub.amount);
    require(success, "Transfer failed");
}
```

By applying this correction, it ensures that the SubExecutor contract handles transactions efficiently and securely, adhering to ERC20 token standards and providing a smooth and reliable user experience in the subscription ecosystem. This adjustment is crucial to prevent service disruptions and ensure the integrity of user funds.

RESULT

```
[PASS] test_FuzzERC20Payment(uint256,uint256) (runs: 256, μ: 572323, ~:
572323)
Traces:
  [572323]
SFoundrySubExecutorTest::test_FuzzERC20Payment(171474132253526330804
[1.714e20], 65)
    ├─ [0] VM::assume(true) [staticcall]
    │   └─ ← ()
    ├─ [0] VM::assume(true) [staticcall]
    │   └─ ← ()
    ├─ [0] VM::startPrank(deployer:
[0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946])
    │   └─ ← ()
    ├─ [29531] MockERC20::mint(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264], 171474132253526330804
[1.714e20])
    │   ├─ emit Transfer(from: 0x0000000000000000000000000000000000000000,
to: SubExecutor: [0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264], value:
171474132253526330804 [1.714e20])
    │   └─ ← ()
    ├─ [0] VM::stopPrank()
```

```
    │   └─ ← ()
    ├─ [0] VM::startPrank(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
    │   └─ ← ()
    ├─ [24658] MockERC20::approve(Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], 171474132253526330804
[1.714e20])
    │   ├─ emit Approval(owner: SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264], spender: Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], value: 171474132253526330804
[1.714e20])
    │   └─ ← true
    ├─ [0] VM::stopPrank()
    │   └─ ← ()
    ├─ [582] MockERC20::balanceOf(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264]) [staticcall]
    │   └─ ← 171474132253526330804 [1.714e20]
    ├─ [2582] MockERC20::balanceOf(Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b]) [staticcall]
    │   └─ ← 0
    ├─ [2556] SubExecutor::getOwner() [staticcall]
    │   └─ ← 0x0000000000000000000000000000000000000000
    ├─ [0] VM::startPrank(0x0000000000000000000000000000000000000000)
    │   └─ ← ()
    ├─ [361573] SubExecutor::createSubscription(Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], 65, 2592000 [2.592e6],
7776001 [7.776e6], MockERC20:
[0xfF2Bd636B9Fc89645C2D336aeaDE2E4AbaFe1eA5])
    │   ├─ [200320] Initiator::registerSubscription(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264], 65, 7776001 [7.776e6],
2592000 [2.592e6], MockERC20:
[0xfF2Bd636B9Fc89645C2D336aeaDE2E4AbaFe1eA5])
    │   │   └─ ← ()
    │   ├─ emit subscriptionCreated(_initiator:
0x0000000000000000000000000000000000000000, _subscriber: Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], _amount: 65)
    │   └─ ← ()
    ├─ [0] VM::stopPrank()
    │   └─ ← ()
    ├─ [0] VM::warp(2592002 [2.592e6])
    │   └─ ← ()
    ├─ [0] VM::prank(Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b])
    │   └─ ← ()
    ├─ [128597] SubExecutor::processPayment()
    │   ├─ [582] MockERC20::balanceOf(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264]) [staticcall]
    │   │   └─ ← 171474132253526330804 [1.714e20]
    │   ├─ [0] console::log("ERC20 Token balance of SubExecutor:",
171474132253526330804 [1.714e20]) [staticcall]
    │   │   └─ ← ()
    │   ├─ [870] MockERC20::allowance(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264], Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b]) [staticcall]
```

```
    │   │   └─ ← 171474132253526330804 [1.714e20]
    │   ├─ [0] console::log("Allowance for Initiator to spend
SubExecutor's tokens:", 171474132253526330804 [1.714e20]) [staticcall]
    │   │   └─ ← ()
    │   ├─ [23010] MockERC20::transfer(Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], 65)
    │   │   ├─ emit Transfer(from: SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264], to: Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], value: 65)
    │   │   └─ ← true
    │   ├─ [0] console::log("ERC20 payment processed from SubExecutor to
Initiator") [staticcall]
    │   │   └─ ← ()
    │   ├─ emit paymentProcessed(_subscriber: Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], _amount: 65)
    │   └─ ← ()
    ├─ [582] MockERC20::balanceOf(SubExecutor:
[0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264]) [staticcall]
    │   └─ ← 171474132253526330739 [1.714e20]
    ├─ [582] MockERC20::balanceOf(Initiator:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b]) [staticcall]
    │   └─ ← 65
    └─ ← ()
```

```
console::log("ERC20 Token balance of SubExecutor:", 171474132253526330804
console::log("Allowance for Initiator to spend SubExecutor's tokens:",
171474132253526330804
```

TEAM RESPONSE

(To be filled with client's response)

7. Incomplete Subscriber Removal Logic in Initiator Contract

SEVERITY

LOW

DESCRIPTION

The Initiator contract maintains a list of subscribers in two structures: a mapping for subscription details (subscriptionBySubscriber) and an array of subscriber addresses (subscribers). While the registerSubscription function correctly adds new subscribers to both structures, the removeSubscription function only removes a subscriber's details from the mapping, neglecting the array. This discrepancy leads to stale data within the subscribers array, potentially causing inaccuracies in subscriber management and enumeration.

LOCATION OF AFFECTED CODE

The issue is located in the removeSubscription function of the Initiator contract:

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

POC

FOUNDRY

```
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 subscribers array");
}
```

RESULT

```
[FAIL. Reason: assertion failed] testRemoveSubscription() (gas: 179578)
Logs:
  Error: Subscriber should be removed from the subscribers array
  Error: Assertion Failed

Traces:
```

```
  [183791] HalmosInitiatorTest::testRemoveSubscription()
    ├─ [0] VM::prank(0x0000000000000000000000000000000000000001)
    │   └─ ← ()
    ├─ [200319]
Initiator::registerSubscription(0x0000000000000000000000000000000000000001
, 100000000000000000 [1e18], 2592001 [2.592e6], 864000 [8.64e5], ERC20:
[0x2e234DAe75C793f67A35089C9d99245E1C58470b])
    │   └─ ← ()
    ├─ [0] VM::prank(0x0000000000000000000000000000000000000001)
    │   └─ ← ()
    ├─ [1664]
Initiator::removeSubscription(0x0000000000000000000000000000000000000001)
    │   └─ ← ()
    ├─ [1909]
Initiator::getSubscription(0x0000000000000000000000000000000000000001)
[staticcall]
    │   └─ ← SubStorage({ amount: 0, validUntil: 0, validAfter: 0,
paymentInterval: 0, subscriber:
0x0000000000000000000000000000000000000000, initiator:
0x0000000000000000000000000000000000000000, erc20TokensValid: false,
erc20Token: 0x0000000000000000000000000000000000000000 })
    ├─ [996] Initiator::getSubscribers() [staticcall]
    │   └─ ← [0x0000000000000000000000000000000000000001]
    ├─ emit log_named_string(key: "Error", val: "Subscriber should be
removed from the subscribers array")
    ├─ emit log(val: "Error: Assertion Failed")
    ├─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
0x6661696c6564000000000000000000000000000000000000000000000000000000,
0x0000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← ()
    └─ ← ()
```

RECOMMENDATION

To ensure consistency between the subscriptionBySubscriber mapping and the subscribers array, implement a mechanism to remove the subscriber from the array upon calling removeSubscription.

This could involve iterating over the array to find the index of the subscriber to be removed, deleting the subscriber from the array, and then shifting the remaining elements to fill the gap, or swapping the last element with the one to be removed for efficiency.

Here's an example approach using the swap-and-pop technique, which is more gas efficient but does not preserve the order of the array:

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

    for (uint i = 0; i < subscribers.length; i++) {
```

```
            if (subscribers[i] == _subscriber) {
                subscribers[i] = subscribers[subscribers.length - 1];
                subscribers.pop();
                break;
            }
        }
    }
```

This adjustment ensures that the subscribers array accurately reflects the current set of active subscribers, aligning with the state of the subscriptionBySubscriber mapping.

TEAM RESPONSE

=== INFORMATIONAL ===

8. Improvement on `processPayment()` function

SEVERITY

INFORMATIONAL

DESCRIPTION

The current implementation of processPayment() function in the SubExecutor contract utilizes msg.sender to identify the subscription initiator. This approach, while functional, may limit the contract's flexibility and clarity regarding subscription management.

Specifically, it relies on the assumption that the msg.sender is always the initiator of the subscription, which could be restrictive in scenarios where third-party contracts or addresses are authorized to manage subscriptions on behalf of initiators.

LOCATION OF AFFECTED CODE

The processPayment() function within the SubExecutor contract is the focus of this informational finding. The function signature does not currently accept any arguments and implicitly uses msg.sender to reference the initiator's subscription:

```
function processPayment() external nonReentrant {
    SubStorage storage sub = getKernelStorage().subscriptions[msg.sender];
    ...
}
```

RECOMMENDATION

To enhance clarity, flexibility, and control over subscription payment processing, it is recommended to modify the processPayment() function to explicitly accept an address _initiator argument. This change would make the intended initiator for payment processing explicit, improving code readability and potentially facilitating more complex subscription management scenarios.

Proposed modification:

```
function processPayment(address _initiator) external nonReentrant {
    SubStorage storage sub = getKernelStorage().subscriptions[_initiator];

    require(block.timestamp >= sub.validAfter, "Subscription not yet
valid");
    require(block.timestamp <= sub.validUntil, "Subscription expired");
    require(msg.sender == sub.initiator || msg.sender == address(this),
"Not authorized");
```

Benefits of this recommendation:

- **Improved Clarity:** Making the initiator explicit in the function call clarifies the intent and operation of the contract, easing maintenance and auditing.
- **Enhanced Flexibility:** This change allows for the possibility of third-party contracts or addresses being permitted to process payments on behalf of the initiator, under controlled circumstances.
- **Reinforced Security:** By explicitly requiring the initiator's address as an argument, the contract can implement more robust authorization checks, ensuring that only authorized entities can process payments.

TEAM RESPONSE

(To be filled with client's response)