MafiaTakeDown Security Audit Report by MK (15th DEC 2024)

High Risk Findings

[H-1] Incorrect dependecies returned in Laundrette policy

Description: `Laundrette::configureDependencies` returns an array of it's Module dependecies incorrectly.

Impact: Module upgrades for "MONEY" keycode modules are not correctly done for the Laundrette policy, meaning that the EmergencyMigration script doesnt actually change the `MoneyShelf` module to the `MoneyVault` module.

Proof of Concept:

1. When the policy is forst activated `configureDependencies` is called. In Laundrette.sol the dependencies are returned incorrectly because the "WEAPN" keycode overwrites the "MONEY" keycode in the position 0 of the dependencies array, this means that during the policy activation this module dependency will not be saved in the `moduleDependents` mapping in the kernel. This affects the functionality of the `_reconfigurePolicies` function which is executed when upgrading modules.

2. Test:

```
function testCanWithdrawAfterMigration() public {
    // Get 2 gangmembers in
    vm.prank(kernel.admin());
    kernel.grantRole(Role.wrap("gangmember"), godFather);
    vm.startPrank(godFather);
    laundrette.addToTheGang(gm1);
    laundrette.addToTheGang(gm2);
    vm.stopPrank();

    // Non gangmember donates money
    vm.startPrank(godFather);
    usdc.transfer(donor, 100e6);
    usdc.transfer(gm1, 100e6);
    usdc.transfer(gm2, 100e6);
    usdc.approve(address(moneyShelf), 100e6);
    vm.stopPrank();

    // Donor deposits money
    vm.prank(donor);
    usdc.approve(address(moneyShelf), 100e6);
    laundrette.depositTheCrimeMoneyInATM(donor, donor, 100e6);
    // Gangmember 1 deposits money
    vm.prank(gm1);
    usdc.approve(address(moneyShelf), 100e6);
    laundrette.depositTheCrimeMoneyInATM(gm1, gm1, 100e6);
```

```
    // EMERGENCY MIGRATION
    vm.startBroadcast(kernel.executor());
    MoneyVault moneyVault = new MoneyVault(kernel, usdc, crimeMoney);
    kernel.executeAction(Actions.UpgradeModule, address(moneyVault));
    vm.stopBroadcast();

    // Gangmember 2 deposits money
    vm.prank(gm2);
    usdc.approve(address(moneyShelf), 100e6);
    vm.expectRevert();
    laundrette.depositTheCrimeMoneyInATM(gm2, gm2, 100e6);
  }
```

3. The last call to the `depositTheCrimeMoneyInATM` function should revert but it doesn't because it's still calling the MoneyShelf contract instead of MoneyVault.

Recommended Mitigation: Change line [26] from: `dependencies[0] = toKeycode("WEAPN");` to `dependencies[1] = toKeycode("WEAPN");`

[H-2] Insecure access control in `Laundrette::depositTheCrimeMoneyInATM` function,, causing anyone can move USDC in other's wallet

Description: The function `Laundrette::depositTheCrimeMoneyInATM` allows an external caller to set arbitrary `account` parameter. And USDC will be transferred from this `account` address. User may lose USDC beacuse of this.

Impact: An attacker can call this function and set `account` to be any address with remaining USDC allowance, and set `to` to be himself (or anyone's address), to steal USDC from `account` address. The attacker can later withdraw USDC by burning crimeMoney.

Proof of Concept:

```
  function test_depositOtherUSDC() public {
      address alice = makeAddr("alice");
      address bob = makeAddr("bob");

      vm.prank(godFather);
      usdc.transfer(alice, 100e6);
      vm.prank(alice);
      usdc.approve(address(moneyShelf), 100e6);

      joinGangGodFather();
      joinGang(bob);

      vm.prank(bob);
      laundrette.depositTheCrimeMoneyInATM(alice, bob, 100e6);
      assertEq(usdc.balanceOf(alice), 0);
      assertEq(usdc.balanceOf(address(moneyShelf)), 100e6);
```

```
        assertEq(crimeMoney.balanceOf(bob), 100e6);
    }
```

Recommended Mitigation:

1. ensure the caller is same as `account` parameter, revert otherwise

```
    function depositTheCrimeMoneyInATM(address account, address to, uint256
amount) external {
++   if (!(account == msg.sender) {
++           revert("Laundrette: you are not authorized to call this function");
++       }
        moneyShelf.depositUSDC(account, to, amount);
    }
```

[H-3] GodFather unable to remove USDC from MoneyVault

Description: If the patron saint of mobsters is feeling kind and by some miracle there is all the USDC in the MoneyVault, the GodFather faces yet more challenges. The withdrawal of funds from the Vault is predicated on the logic within the `withdrawUSDC` function, and sadly, this is identical to the MoneyShelf contract which checks the bank mapping balance for the calling account as well as requiring the burning of CrimeTokens, neither of which happens to have favourable values for the GodFather.

Impact: The USDC is locked up without the correct balance in the bank mapping and the right amount of CrimeTokens.

Proof of Concept:

1. The MoneyVault contract has the same logic as the MoneyShelf when it comes to withdrawals. It burns an equivalent amount of CrimeTokens to release the USDC, as well as reducing the balance stored. Given the balance isn't even brought over, this is set to zero for the GodFather in the MoneyVault. This means the amount of USDC released is rather limited. Every attempt to remove any USDC that miraculously works its way into MoneyVault will result in the inevitable `panic: arithmetic underflow or overflow` errors.

2. forge test to demonstrate (along with some help as this situation requires massaging to get to the situation where there is actual USDC in MoneyVault.

```
    // This assumes we have fixed the laundrette dependency issue somehow
    function test_EmergencyMigrationSuccess() public {
        joinGang(address(this));

        // Godfather deposits his USDC into the ATM
        vm.startPrank(godFather);
        usdc.approve(address(moneyShelf), 1e6);
        laundrette.depositTheCrimeMoneyInATM(godFather, godFather, 1e6);
        vm.stopPrank();
```

```
        // Check we are still using MoneyShelf
        assertEq(address(kernel.getModuleForKeycode(Keycode.wrap("MONEY"))),
address(moneyShelf));

        // Raid time. Godfather pulls the plug.
        EmergencyMigration migration = new EmergencyMigration();
        MoneyVault moneyVault = migration.migrate(kernel, usdc, crimeMoney);

        // Check we are now using MoneyVault
        assertNotEq(address(moneyShelf), address(moneyVault));
        assertEq(address(kernel.getModuleForKeycode(Keycode.wrap("MONEY"))),
address(moneyVault));

        // Check to see where the USDC actually is
        assertEq(usdc.balanceOf(address(moneyVault)), 0);
        assertEq(usdc.balanceOf(address(moneyShelf)), 1e6);

        // Now massage the numbers to make it look like the migration was
successful
        vm.startPrank(godFather);
        usdc.transfer(address(moneyVault), 10e6);
        vm.stopPrank();

        assertEq(usdc.balanceOf(address(moneyVault)), 10e6);
        console.log("Godfather balance of USDC: ",
usdc.balanceOf(address(godFather))); //999989000000

        // try and get USDC out of the vault
        vm.startPrank(godFather);
        laundrette.withdrawMoney(godFather, godFather, 1e6);

    }
```

Recommended Mitigation:

1. However if the GodFather has some coding-fu, then they could deploy a new MoneyVault contract with a proper init which removes the withdraw function and CrimeMoney.burn function in withdrawUSDC.

2. To pay homage and have at least some consistency with the framework itself, a `permissioned` modifier should be added. There is a check for the executor already, and in this situation, engineering niceness is probably not high on the list of priorities but it would show a degree of class hitherto not evident within this organisation.

```
-    function withdrawUSDC(address account, address to, uint256 amount) external {
+    function withdrawUSDC(address account, address to, uint256 amount) external
permissioned {
        require(to == kernel.executor(), "MoneyVault: only GodFather can receive
USDC");
-        withdraw(account, amount);
-        crimeMoney.burn(account, amount);
```

```
        usdc.transfer(to, amount);
    }
```

Medium Risk Findings

[M-1] Any gangmember can remove gangmember role for any other member

Description: Any user with role "gangmember" can call the function `quitTheGang` passing the account of any other gangmember

Impact: Denial of service, malicious user can remove users from gang, including the godFather, making godFather unable to use some of the functionality.

Proof of Concept:

1. Any user with gangmenber role can keep forcing to quit other gangmember users due to a lack of access control in the `quitGang` function.

Foundry test:

```
function testOtherGangMembersCanQuitYou() public {
    vm.prank(kernel.admin());
    kernel.grantRole(Role.wrap("gangmember"), godFather);
    vm.prank(godFather);
    laundrette.addToTheGang(gm1);
    vm.prank(gm1);
    laundrette.quitTheGang(godFather);
    vm.prank(godFather);
    vm.expectRevert();
    laundrette.addToTheGang(gm2);
  }
```

Recommended Mitigation:

1. Add a check to verify that the account being passed is the same as msg.sender, or depending on the expected functionality add the `isAuthorizedOrRevert` modifier to the `quitTheGang` function.

[M-2] Potential Risks in Handling USDC within Protocol (Blacklists, Low Decimals, Transfer Fees)

Description: The Mafia Protocol, as detailed in the provided README, may encounter significant issues when interacting with USDC due to potential blocklists, low decimals, and transfer fees. These vulnerabilities could lead to the protocol being compromised or funds being trapped, affecting the security and functionality of the protocol.

Impact: These vulnerabilities could lead to:

1. Trapped Funds: Blocklisting the contract address can make all USDC funds irretrievable. (The anti-mafia agency could use this to prevent withdrawals)
2. Precision Loss: Low decimals can result in incorrect balances and value calculations. (The protocol doesn't perform any advanced calculations at this point)
3. Unexpected Deductions: Transfer fees can disrupt transaction integrity, leading to potential financial loss. (If the USDC contract gets upgraded on mainnet for example)

Proof of Concept:

1. Tokens with Blocklists Some tokens, such as USDC, have an admin-controlled address blocklist. If the contract address gets added to this blocklist, transfers to and from the address are prohibited. This scenario can arise due to regulatory actions, malicious behavior, or compromised token owners, resulting in trapped funds within the contract.

2. Low Decimals USDC has a low decimal count (6), and other tokens can have even fewer decimals. This can cause unexpected precision loss during calculations and transfers, potentially leading to significant discrepancies in token balances and values within the protocol. The protocol also implemented a `MockUSDC` contract for local testing but the mocked contract doesn't have the correct token decimal (6) and instead uses the default (18) which could cause problems if not handled carefully.

3. Fee on Transfer Certain tokens impose transfer fees. While USDC does not currently charge a fee, it may implement one in the future. This behavior can lead to unanticipated deductions from token transfers, disrupting expected transaction amounts and protocol functionality.

Recommended Mitigation: Blocklist Mitigation:

- Use multiple token options and fallback mechanisms.

Transfer Fee Consideration:

- Check for the `transferFee` parameter in token contracts.
- Implement checks and balances to account for potential transfer fees.

Decimal Handling:

- Change the `MockUSDC` contract to

```
    contract MockUSDC is ERC20 {
        constructor() ERC20("USDC", "USDC") {
            _mint(msg.sender, 1_000_000e6);
        }

+       function decimals() public view virtual override returns (uint8) {
+           return 6;
+       }
}
```

[M-3] Lack of Fund Transfer in `EmergencyMigration::migrate` Function Compromises Security of Emergency Migration

Description: The `EmergencyMigration::migrate` doesn't transfer the funds from the MoneyShelf contract to the MoneyVault contract

Impact: Since the funds are not transferred to the MoneyVault, they remain in the MoneyShelf and are susceptible to being compromised.

Proof of Concept:

1. The goat of the `EmergencyMigration::migrate` is to allow the godfather to migrate funds from the MoneyShelf to a contract (MoneyVault) that only he can manage. However, the function currently does not transfer the funds, thus defeating the purpose of the emergency migration
2. Place the following into `EmergencyMigration.t.sol`.

```
    function test_funds_are_not_transfered() public {
        vm.prank(godFather);
        usdc.transfer(address(this), 100e6);
        usdc.approve(address(moneyShelf), 100e6);
        laundrette.depositTheCrimeMoneyInATM(address(this), address(this), 100e6);
        assertEq(usdc.balanceOf(address(this)), 0);
        assertEq(usdc.balanceOf(address(moneyShelf)), 100e6);
        console.log("moneyshelf balance: ", usdc.balanceOf(address(moneyShelf)));
// MoneyShelf has 100 USDC

        assertEq(address(kernel.getModuleForKeycode(Keycode.wrap("MONEY"))),
address(moneyShelf));

        EmergencyMigration migration = new EmergencyMigration();
        MoneyVault moneyVault = migration.migrate(kernel, usdc, crimeMoney);

        assertNotEq(address(moneyShelf), address(moneyVault));
        assertEq(address(kernel.getModuleForKeycode(Keycode.wrap("MONEY"))),
address(moneyVault));
        console.log("moneyVault balance:", usdc.balanceOf(address(moneyVault)));
// MoneyVault has 0 USDC
    }
```

Recommended Mitigation:

1. Add a transfer of funds from the MoneyShelf to the MoneyVault in the `migrate` function and keep track of balance of every account.

[M-4] `MoneyVault::Withdraw` Reverts Due to Restrictive onlyMoneyShelf Modifier in `CrimeMoney::burn` Function

Description: The `CrimeMoney::burn` function is restricted by an onlyMoneyShelf modifier, meaning that only the MoneyShelf contract can burn CrimeMoney. However, the burn function is also called in the `MoneyVault::withdraw` function, causing it to revert due to the lack of the necessary role being granted to the MoneyVault in the `EmergencyMigration.s.sol` script.

Impact: The restriction causes the withdrawal functionality of the MoneyVault to be inoperative, effectively trapping all funds within the MoneyVault contract. This can lead to significant operational issues and loss of access to funds.

Proof of Concept: The `MoneyVault::withdraw` function relies on the `CrimeMoney::burn` function, which is currently restricted by the onlyMoneyShelf modifier. This modifier only allows the MoneyShelf contract to burn CrimeMoney. As the MoneyVault does not have this role, any attempt to withdraw from the MoneyVault will revert. Additionally, the revert message "CrimeMoney: only MoneyShelf can mint" is misleading when used in the context of the burn function.

Recommended Mitigation:

1. Add a new modifier onlyAuthorized

```
modifier onlyAuthorized() {
    require(
        kernel.hasRole(msg.sender, Role.wrap("moneyshelf")) ||
kernel.hasRole(msg.sender, Role.wrap("moneyvault")),
        "CrimeMoney: only MoneyShelf or MoneyVault can perform this action"
    );
    _;
}
```

2. Update the CrimeMoney::burn function to use the onlyAuthorized modifier:

```
- function burn(address from, uint256 amount) public onlyMoneyShelf {
+ function burn(address from, uint256 amount) external onlyAuthorized {
    _burn(from, amount);
}
```

Low Risk Findings

[L-1] Function to retrieve admin status does not work due to missing privileges

Description: The `Laundrette` contract provides a function `retrieveAdmin()` which is meant to be used by the godfather to gain admin status in the system. This can be necessary when the godfather needs the privileges to grant and revoke roles, as only the `Kernel::admin` is allowed to do that. However, `Laundrette::retrieveAdmin` is not able to change the admin because it doesn't have the **executor** status.

Impact: There's no serious impact here because the only thing that happens is that the function will always revert. The godfather can just call `Kernel::executeAction(Actions.ChangeAdmin, Kernel::executor)` directly and gains admin status like that.

Proof of Concept:

1. There are types of roles in the mafia takedown contracts:
    - Built-in roles, of which there are `Kernel::admin` and `Kernel::executor`
    - Custom roles, which are created via `Kernel::grantRole`
2. What's important to note here, is that only `Kernel::admin` is able to grant and revoke roles. `Kernel::executor` on the other hand, has the ability to execute actions via `Kernel::executeAction`, such as `Actions.InstallModule`, `Actions.ActivatePolicy`, but also actions related to built-in roles, such as `Actions.ChangeAdmin` and `Actions.ChangeExecutor`.
3. When the system is deployed, the `Laundrette` contracts becomes the admin while the godfather becomes the executor:

```
kernel.executeAction(Actions.ChangeAdmin, address(laundrette));
kernel.executeAction(Actions.ChangeExecutor, godFather);
```

4. This means, `Laundrette` can grant and revoke roles, `godFather` can execute actions.

The `Laundrette::retrieveAdmin` function, which is meant to be used by the godfather to gain admin status, makes use of `Kernel::executeAction` to change the admin:

```
function retrieveAdmin() external {
    kernel.executeAction(Actions.ChangeAdmin, kernel.executor());
}
```

5. As we've learned however, `Laundrette` has to be the executor to call `Kernel::executeAction`. Only the godfather can call `Kernel::executeAction`.
6. This function will revert no matter who calls it.

Recommended Mitigation:

1. For this to work, `Laundrette` would need to become the executor as well. In addition, the account used in `Kernel::executeAction` needs to be parametrized. The godfather can then use the `ChangeExecutor` action to make `Laundrette` the executor, which would make `Laundrette::retrieveAdmin` work again.
2. However, **this is not recommended** because at that point, `Laundrette` would be the executor and there's no way to give that status to a different account. In addition, this would require additional access control though because that would allow anyone to call this function with any account to become a new admin.
3. Instead, I'd recommend to just remove `Laundrette::retrieveAdmin` altogether and simply let the executor call `Kernel::executeAction` directly to make any changes to `Kernel::admin`.

```
- function retrieveAdmin() external {
-     kernel.executeAction(Actions.ChangeAdmin, kernel.executor());
```

```
- }
```

4. Below is a test that can be added to `Laundrette.t.sol` to verify that `Laundrette::retrieveAdmin` always fails:

```
+ function test_retrieveAdminBug() public {
+    vm.prank(godFather);
+    vm.expectRevert(abi.encodeWithSignature("Kernel_OnlyExecutor(address)",
laundrette));
+    laundrette.retrieveAdmin();
+ }
```

[L-2] Godfather is not initially added to the gang members

Description: Godfather is not initially added to the gang members which prevents him from calling some of the `Laundrette.sol` functions

Impact: Unnecessary gas will be spent.

Proof of Concept:

1. All the functions in `Laundrette.sol` which have the `onlyRole("gangmember")` modifier can't be called by the godfather as they will revert, because he doesn't have the gangmember role.
2. There is a way around this, but not without a cost:

```
1. the godfather will have to revoke the admin rights from `Laundrette.sol` to
himself

2. then call `Kernel::grantTole`

3. give the admin rights back to `Laundrette.sol`.
```

3. Making these 3 transactions will be gas costly.
4. To test the result paste these tests in Laundrette.t.sol:

```
function test_godFatherCannotWithdraw() public {
        vm.prank(godFather);
        usdc.approve(address(moneyShelf), 100e6);
        laundrette.depositTheCrimeMoneyInATM(godFather, godFather, 100e6);
        assertEq(usdc.balanceOf(address(moneyShelf)), 100e6);
        assertEq(crimeMoney.balanceOf(godFather), 100e6);

        vm.expectRevert();
        laundrette.withdrawMoney(godFather, godFather, 100e6);
```

```
    }

    function test_godFatherCannotAddToGang() public {
        vm.prank(godFather);

        vm.expectRevert();
        laundrette.addToTheGang(address(this));
    }
```

Recommended Mitigation:

1. Grant the gangmember role to the godfather in the `Deployer.s.sol` contract:

```
function deploy() public returns (Kernel, IERC20, CrimeMoney, WeaponShelf,
MoneyShelf, Laundrette) {
        godFather = msg.sender;

        // Deploy USDC mock
        HelperConfig helperConfig = new HelperConfig();
        IERC20 usdc = IERC20(helperConfig.getActiveNetworkConfig().usdc);

        Kernel kernel = new Kernel();
        CrimeMoney crimeMoney = new CrimeMoney(kernel);

        WeaponShelf weaponShelf = new WeaponShelf(kernel);
        MoneyShelf moneyShelf = new MoneyShelf(kernel, usdc, crimeMoney);
        Laundrette laundrette = new Laundrette(kernel);

        kernel.grantRole(Role.wrap("moneyshelf"), address(moneyShelf));
+       kernel.grantRole(Role.wrap("gangmember"), address(godfather));
```

[L-3] `crimeMoney` not pegged to USDC

Description: value deposited/withdrawn different from that minted/burnt because USDC has only 6 decimals and crimeMoney has 18

Impact: Not in sync with the documentation, as it wwas stated that `crimeMoney` is pegged to USDC but it is not; If any user deposits 1,000,000,000$ worth of USDC, they're getting 1000/(10**12) so 0.001$ worth of `crimeMoney`

Proof of Concept: `crimeMoney` is overcollateralized: the amount of usdc provided to keep that peg is much more than it should be (10**18/10**6 = 10**12)

Recommended Mitigation:

1. The amount of crimeMoney minted during the deposit should be multiplied by 10**12
2. The amount of crimeMoney burnt during the deposit should be divided by 10**12