Firm

Firm Protocol Audit

coinspect

Firm Protocol

Smart Contract Audit

V230222

Prepared for Firm Protocol • February 2023

1. Executive Summary

In **January 2023**, **Firm** engaged Coinspect to perform a source code review of **Firm Protocol**. Firm Protocol is a provider of internet-native business solutions seeking to replace legal documents and paperwork via a modular design. Firm brings an architecture that offers customization options for clients to choose between the modules they require. Thanks to its modular design, new modules could be added in the future bringing new functionalities according to the current industry needs. The objective of the project was to evaluate the security of the smart contracts.

The following issues were identified during the initial assessment:

High Risk	Medium Risk	Low Risk
Open **0**	Open **0**	Open **0**
Fixed 2	Fixed 5	Fixed 0
Reported 3	Reported 6	Reported 3

All of the high risk issues (**FRM-01**, **FRM-02**, **FRM-03**) are related to `Captable` and describe ways in which a company can steal from vested employees and censor voting proposals, as well as a way in which the whole voting system can be bricked. **FRM-09** is also related to voting, describing how it can lead to a company takeover. **FRM-05** shows a manager locking a user from their vesting. **FRM-04** shows the risk of replay for users after using the relayer. **FRM-06** describes how easy it is to give admin rights to users by mistake. **FRM-08** describes an attack that would waste relayer's funds, while **FRM-07** shows misaccounting risks when using fee-on-transfer tokens. **FRM-10**, **FRM-11** and **FRM-12** are low risk issues that do not pose an immediate threat.

2. Assessment and Scope

Firm Protocol brings a set of modules that work on top of a Gnosis Safe, which allows the integration of external modules. Via this mechanism, Firm offers customizable modules with features that companies frequently require such as budget handling, access control, equity distribution and voting.

The audit started on **January 30, 2023** and was conducted on the tag `coinspect-audit` of the git repository at https://github.com/firm-org/firm-protocol/ as of commit `7186feb3abba6666e7f115ffe56264d1d76f4dfb` of Jan 27, 2023.

The audited files have the following sha256sum hash:

```
87840962f9864d3b2760eb0b1c2f5e0951e79a1588de4441501753ea69fb09fe  ./captable/controllers/VestingController.sol
b8b2530ac4085fe6fee2f963108e331a7e894687aebb4840c631be8575f36620  ./captable/controllers/AccountController.sol
2d5295bd5da772b0d7fb660ef55145ae4253984558192e373351c1640f00c2a5  ./captable/interfaces/IBouncer.sol
fefa6214332b1437fce305165c11da751d28beb01888c97740718a69d9f599c1  ./captable/interfaces/ICaptableVotes.sol
eeefebac55d4a398e6432a27c426d670720be2bad7e5fe608261b670cdffe082  ./captable/EquityToken.sol
a1c5e99214a40565d9d655fdf3a1f120dfa350585506b0ccca2a98c35cb4dbbe  ./captable/BouncerChecker.sol
45f9bcf115cfd793543ae8e9a802ece4e06ee664c77af59df6db77e78cafd81a  ./captable/Captable.sol
9a7fd3ce6f25ddaa1e57bdc9ffc2ff4b5da03a475c4b76b3b8782e34dddda785  ./voting/OZGovernor.sol
e0793e402eca346384d0b88a0616846d235f5965376880a63081daab8865a07a  ./voting/lib/GovernorCaptableVotes.sol
95d1436f3758b7c12bd8f65a7fc8b8e99fa92b212d42cc358a5da032cf637d7f
./voting/lib/GovernorCaptableVotesQuorumFraction.sol
9059c4908e3884361aaa8d91851016875c6506e43b2bab64746234827158dbfd  ./voting/Voting.sol
e933e0e76461457bffd4c50460aaaf1e6d17e9a0a08edde1552eb749a7ff8c46  ./budget/modules/BudgetModule.sol
1894ff03dc169211b17cbc59b7b8c77a3f5c4609fe2573451a2bf3dc7a8fe1e4  ./budget/modules/streams/ForwarderLib.sol
1ac5ba250246ff1e70392ac4338080fec2ad3bc8e0ed3c5c9eb038df572e9849  ./budget/modules/streams/LlamaPayStreams.sol
76d869734c84ab12125ee613c3cb72118832b79f98779ecb17de4b635527da2c  ./budget/Budget.sol
2ece7c0914a2a9f0bf765b27d52a38fb1c1ff1aaf587a96f6e39375c278e921a  ./budget/TimeShiftLib.sol
826d21e41ace4e9fb0f0e6ed3294c9828a6d1e1e6046dcdb809d5b9211019dd7  ./metatx/FirmRelayer.sol
1241809e6fb3e7e90b4442c2990483c1dbb88b8118f53dc1ecaa3fcbb78c2964  ./bases/utils/AddressUint8FlagsLib.sol
a930f40790076832efeadc6aef0ef74b56f6cffe030a24f05652fbe0def87589  ./bases/RolesAuth.sol
f4ce45e6b5c2f872d989102f4e3338239e8697d56a94e4b4dbc32841d8d68b03  ./bases/ERC2771Context.sol
7eb962c8d1a801b2c355c890fe6f00e8df3f7eda133ba742b22aace81ae63ac6  ./bases/interfaces/IModuleMetadata.sol
105e6674a840fe7968ebcc2638e5395de69ec7db92fcc1fff35dfcf12516bbda  ./bases/interfaces/ISafe.sol
4b70406bdc6cf8e56601e04b7497ddd1840b017b42fd9217e77f51b58cf904a0  ./bases/EIP1967Upgradeable.sol
4592506c82cbd3c1ea87b97117f701285722f217174cfd94791970e287f37e01  ./bases/FirmBase.sol
a33878fd7af24f52e00c0812e65493a7ea7936a9d8ccd03a0ce72228054c604b  ./bases/SafeModule.sol
652f3a932bb3c93a77948d345806f9f9c8d2f31a755c1b4bb1e0740805697139  ./bases/SafeAware.sol
a322c0db68db1664eb62f39849ad35820b381fae522f989d9f98189b960f99f6  ./factory/FirmFactory.sol
1151c99d3651188c6c74da3b7f9ef83e18a6baf7930e1737f31d801637956351  ./factory/UpgradeableModuleProxyFactory.sol
85b153a660ffd2fec31e42f9b9d20e09e39ffeed666dc7abeb36c17e6036f048  ./roles/interfaces/IRoles.sol
22b2f29757099fd7a1041ad9d1c4e282644ea01da140cd044260990a79e1771b  ./roles/Roles.sol
```

During the audit, Coinspect considered Firm intentions for their platform to perform a threat model. The objective of Firm is to be a platform to manage companies and reduce or eliminate the need for bureaucracy in an organization.

As the usage of Firm's modules was within a company, Coinspect focused on ways to break its hierarchy, looking for ways in which an employee could impact the company or a superior. Conversely, problems that need to be triggered by the Vault

or managers were generally deemphasised. These assumptions do not hold everywhere: some modules are *intended* to be a protection for employees and they have to make sure managers and the company itself do not abuse their power. That is the case of, for example, the Vesting controller.

Coinspect considered not only the security of the product against active attackers, but also the general safety for end users, whether they are company board members, managers, or employees. Coinspect found the risk of misusing modules to be a real threat. This is reflected in two main issues: one regarding voting proposals which can lead to total destruction or drainage of the system; the other dealing with admin permissions over roles which can unexpectedly make users admins of roles.

It is worth noting that some of the issues documented are considered intended behavior by Firm. Although Coinspect understands the need for flexibility when creating a system intended to be somehow adapted to different end-user needs, the risks are there. For each issue, Coinspect provided tests that clearly showcase the dangers posed and proposed mitigations that do not lose any flexibility while being safer for users.

The project was organized and easy to read. It is a Foundry environment where the protocol's contracts are compiled with the 0.8.17 Solidity's version. All the tests ran smoothly and the vast array of tests provided allowed Coinspect to quickly reproduce and run proof of concepts that provide a better illustration of relevant aspects of the code.

Firm provides a modular architecture where every module inherits from base contracts which implement upgradability by default (meaning that for every business this will be an opt-out feature). Access control modifiers are provided to ensure that modules work as intended, and proxies use randomized slots to ensure that no collisions exist. Firm relies on Gnosis Safe as the modules provided are installed thanks to that flexibility provided by Gnosis. The deployment of Firm for every client is handled by a factory that outsources the multisig deployment to the Gnosis Safe Factory. The installation of each Firm's module is made via a callback triggered once the safe was deployed and it is executed by a `delegatecall` reading the factory's logic in the context of the safe. Currently, Firm provides four modules:

Budget, Roles, Captable and Voting; upon deployment only the first two are compulsory making Captable and Voting optative.

The core modules provided have the following key properties and features.

Budget:

- Enables a system of allowance based on a tree where each node can create an arbitrary amount of childs.
- Allows executing payments that **bypass the multisig** performed by authorized actors.
- Brings a recurrency system that enables periodic allowance refreshes (e.g. having a daily recurrent allowance of X, grants the user X per day to be spent).
- Can be topped up with external streaming contracts, such as LlamaPay that enable streamed payments (which could be used to pay for salaries, for example).

Roles:

- Enables a custom role-based access-control mechanism for other core Firm modules.
- Roles that derive from this module cannot impersonate or perform actions on behalf of the safe.
- Has a fixed amount of roles by design. Each role has two different clearances, administrator and non administrator.

Captable:

- Manages ownership and voting rights in a company using Firm via `Equity Tokens`. The goal is to represent shares of stock of legal companies in several jurisdictions via those tokens.
- The safe is allowed to create many `Equity Tokens` (classes) with arbitrary voting weight, total supply and is able to specify a manager for each token. The manager is able to issue those tokens. The safe can modify the maximum amount of mintable tokens and also can freeze a token (which freezes the maximum total supply, the current managers and the active Bouncer). Lastly, an Equity Token can be configured to allow its conversion to a different Equity Token.

- A `Bouncer` could be added to each token, a contract that controls how transfers could be made. Also, this module allows adding a `Controller` to an account holding a certain class of Equity Token (e.g. a Vesting Controller).

Voting:

- A governance module that is wrapped over OpenZeppelin's Governor.
- Tightly related with `Captable` as the voting power and the quorum are calculated considering all the existing `Equity Tokens` supply, balances and weights.
- Proposals that succeed are enabled to trigger an execution. The main difference with regular governance implementations is that the calls executed are triggered in the safe's context using delegatecall.

In addition to the core modules, Firm provides a relayer that enables sponsored transactions. The base modules mentioned before also allow setting several trusted forwarders that can perform relayed calls enabling gasless transactions.

Fixes review

The review of fixes started on February 15th. Firm fixed most of the issues described in this report. The fixes were reviewed in the following pull requests:

- https://github.com/firm-org/firm-protocol/pull/163
- https://github.com/firm-org/firm-protocol/pull/179
- https://github.com/firm-org/firm-protocol/pull/180
- https://github.com/firm-org/firm-protocol/pull/169
- https://github.com/firm-org/firm-protocol/pull/182
- https://github.com/firm-org/firm-protocol/pull/168
- https://github.com/firm-org/firm-protocol/pull/181
- https://github.com/firm-org/firm-protocol/pull/184

The biggest change was the introduction of the Semaphore component to mitigate issues **FRM-02** and **FRM-09**. While Coinspect found that the new component **can** be used to mitigate these issues, it also believes the component is under tested at the moment and is not ready for general use. The component is extremely powerful

and flexible, and it allows contracts to be **permanently locked** if it is used in the wrong way.

The `Semaphore` uses the `Gnosis Guard` system to restrict actions that the Safe can perform either directly through its signers or via voting. Just like Gnosis Guard, it introduces the risk of blocking the contracts and the safe itself. Nevertheless, it allows users to set very particular configurations that limit the power that voters might acquire.

To mitigate the risks introduced by the new component, Coinspect recommends to:

1. Add integration tests for the `Semaphore` component, especially in regards with interaction with `Voting`.
2. Provide extremely detailed documentation on it where its risks are clearly outlined.
3. Go through a *beta testing* phase for willing users.
4. Make the following changes to the Semaphore smart contract:
    a. Only use function signatories to `DENY` calls, as this will avoid problems with potential signature collisions that could execute arbitrary logic.
    b. Provide more preset configurations for the Semaphore to decrease the amount of users that need to manually configure it.
    c. Ensure that always either the Safe, the Voting contract or both have the `Allow DefaultMode` in the Semaphore. Disallowing both of them will lock the contracts and the Safe itself.
    d. Provide getters that allow querying all the exceptions for a certain account.

3. Summary of Findings

Id	Title	Total Risk	Fixed
FRM-01	Vesting can be revoked retroactively	High	✔
FRM-02	Safe can ignore votes	High	!
FRM-03	Overflow bricks the voting system	High	✔
FRM-04	User transactions can be replayed after failure	Medium	✔
FRM-05	Managers are able to lock zero-balance account with vesting controller	Medium	✔
FRM-06	Admin is given to unexpected users	Medium	✔
FRM-07	Fee on transfer tokens grant an excess of allowance	Medium	✔
FRM-08	Attackers are able to waste relayer's money	Medium	✔
FRM-09	Safe can be taken over through votes	Medium	!
FRM-10	Safe accepts money that it does not own	Low	!
FRM-11	Multipayment recipient prevents others from being paid	Low	!
FRM-12	Setting new allowance amount enables front running	Low	!
FRM-13	Captable accepts 129 classes	Info	✔

4. Detailed Findings

FRM-01 Vesting can be revoked retroactively

Total Risk	Impact	Location
High	High	src/captable/controllers/VestingController.sol
Fixed ✔	Likelihood High	

Description

The vesting contract allows an address authorized by the manager to send all the balance of a holder to the vault, even those that have been already earned via vesting or that were received independently from the vesting system.

This makes it extremely easy for managers to promise vesting returns to their users and then take their rewards. It is important to remark that revokers are not necessarily trusted accounts and are set when a new vesting is created.

Test

```
    function test_CannotRevokeRetroactively() public {
      address anotherUser = address(0x99991111);
      uint256 amount = 100;


      VestingController.VestingParams memory vestingParams;
      vestingParams.startDate = 100;
      vestingParams.cliffDate = 150;
      vestingParams.endDate = 500;
      vestingParams.revoker = VESTING_REVOKER_ROLE_FLAG;


      vm.prank(ISSUER);
      captable.issueAndSetController(HOLDER1, classId, amount, vesting,
abi.encode(vestingParams));
      assertEq(token.balanceOf(HOLDER1), amount);
      assertEq(address(captable.controllers(HOLDER1, classId)), address(vesting));


      vm.prank(ISSUER);
      captable.issue(anotherUser, classId, amount);
      assertEq(token.balanceOf(anotherUser), amount);
```

```
        // Some time passes and the holder transfers the available amount
        vm.warp(200);
        uint256 transferrableAmount = amount - amount * (vestingParams.endDate - 200) /
(vestingParams.endDate - vestingParams.startDate);
        vm.prank(HOLDER1);
        token.transfer(address(0x155), transferrableAmount);
        assertEq(token.balanceOf(address(0x155)), transferrableAmount);
        assertEq(token.balanceOf(HOLDER1), amount - transferrableAmount);

        // The holder receives a transfer from another user
        vm.prank(anotherUser);
        token.transfer(HOLDER1, transferrableAmount);
        assertEq(token.balanceOf(HOLDER1), amount);


        vm.warp(480);
        // The revoker takes all the tokens of the Holder retroactively
        vm.prank(address(VESTING_REVOKER));
        vesting.revokeVesting(HOLDER1, classId, 101);


        // The following assertions will succeed if the vesting is not revoked retroactively
        // and will fail if a retroactive revocation could be performed.
        assertGt(token.balanceOf(HOLDER1), 0);
        assertLt(token.balanceOf(address(safe)), amount);
    }
```

Recommendation

Do not allow vesting rewards to be retroactively revoked.

Status

Fixed in commit `80bb625df334fce9dde3b6432a3b3d520829d76b` of `origin/vesting-future-effective-date` branch.

A revert is triggered if the effective date is in the past. A new function was added that revokes the vesting at the current timestamp. In addition, an unreachable condition inside `revokeVesting()` was removed as the `calculateLockedAmount()` function contemplates the scenario where the timestamp is greater than the vesting end date, covering all the possible time ranges.

FRM-02 Safe can ignore votes

Total Risk	Impact	Location
High	High	`src/captable/Captable.sol` `src/voting/Voting.sol`
Fixed !	Likelihood High	

Description

The safe is capable of ignoring any decision made through voting. The safe can override the voting system in several ways, for example by upgrading contracts so that the proposal is not executed or by minting to themselves a new token with enough voting power.

This is worrisome as voting should be a way in which the company gives power to some stakeholders; but this expectation is subverted as the safe can override any proposal while it is in the `VOTING_PERIOD` or `VOTING_DELAY`.

Test

The following example shows how the safe can prevent an undesired proposal to be executed:

```
function test_SafePowerPerpetration() public {
    address NEW_USER = address(0x123456);
    uint256 amount = 100;
    uint256 UNPAID_SALARY_OWED = 10 ether;

    // The safe freezes the classId
    vm.prank(address(safe));
    captable.freeze(classId);

    vm.prank(ISSUER);
    captable.issue(NEW_USER, classId, amount);

    // The user self-delegates those votes
    vm.prank(NEW_USER);
    token.delegate(NEW_USER);
    vm.roll(3);
    assertTrue(captable.getPastVotes(NEW_USER, 2) == 100);

    bytes memory createAllowanceCalldata = abi.encodeCall(Budget.createAllowance, (
        NO_PARENT_ID,
        NEW_USER,
```

```
            NATIVE_ASSET,
            UNPAID_SALARY_OWED,
            TimeShift(TimeShiftLib.TimeUnit.Daily, 0).encode(),
            ""));

        vm.prank(NEW_USER);
        uint256 proposalId = voting.propose(
            arr(address(budget)),
            arr(uint256(0)),
            arr(createAllowanceCalldata),
            "WE ARE NOT GETTING PAID, CREATE ALLOWANCE FOR SALARIES OWED");

        // The safe wants to stop this proposal and creates a new token with more power
        vm.startPrank(address(safe));
        (uint256 newClassId, EquityToken newToken) =
            captable.createClass("Common", "TST-A", INITIAL_AUTHORIZED, NO_CONVERSION_FLAG,
100, ALLOW_ALL_BOUNCER);
        captable.setManager(newClassId, address(safe), true);
        captable.issue(address(safe), newClassId, 2); // 200 votes.
        vm.stopPrank();

        blocktravel(VOTING_DELAY + 1);

        vm.prank(address(safe));
        voting.castVote(proposalId, 0); // votes against

        vm.prank(NEW_USER);
        voting.castVote(proposalId, 1); // vote for

        blocktravel(VOTING_PERIOD);

        vm.expectRevert(bytes("Governor: proposal not successful"));
        voting.execute(
            arr(address(budget)),
            arr(uint256(0)),
            arr(createAllowanceCalldata),
            keccak256(bytes("WE ARE NOT GETTING PAID, CREATE ALLOWANCE FOR SALARIES OWED"))
        );
    }
```

Recommendation

One way in which this could be mitigated would be to implement the recommendations of **FRM-09** and also limit the actions the Safe can take while a voting proposal is active.

Status

Fixed with Pull request 184 which introduces the Semaphore on commit hash e28db7003629e01fc0179ef919c987365858aff0. See the notes on the Fixes review section of the audit.

FRM-03 Overflow bricks the voting system

Total Risk	Impact	Location
High	High	`src/captable/Captable.sol`
Fixed ✔	Likelihood High	

Description

All users will be prevented from using the voting system if the weighted total supply of all equity tokens overflows.

This is caused by an addition and multiplication in the `_weightedSumAllClasses` method, which is used to calculate several important values for voting such as the quorum needed for a vote.

```
function _weightedSumAllClasses(bytes memory data) internal view returns (uint256 total) {
    uint256 n = classCount;
    for (uint256 i = 0; i < n;) {
        Class storage class = classes[i];
        uint256 votingWeight = class.votingWeight;
        if (votingWeight > 0) {
            (bool ok, bytes memory returnData) = address(class.token).staticcall(data);
            require(ok && returnData.length == 32);
            total += votingWeight * abi.decode(returnData, (uint256));
        }
        unchecked {
            i++;
        }
    }
}
```

It is feasible for this calculation to overflow, both intentionally and unintentionally. The data being fetched from the `staticcall` varies, but when calculating the quorum it is the `totalSupply()` of the token. Unintentionally, it might just be the case that a company has a very high liquidity token with voting weight. Intentionally, it might be a malicious manager with permissions to emit a token with high `maxSupply()` and a certain voting weight.

Test

```
test_VotesShouldNotBeBrickedByIssuance(uint256 maxAuthorized, uint64 votingWeight) public {
    vm.assume(votingWeight > type(uint64).max / 4);
```

```
        vm.assume(maxAuthorized < type(uint224).max);
        vm.assume(maxAuthorized > 0);

        address NEW_USER = address(0x12345);

        vm.startPrank(address(safe));
        (uint256 classId2, EquityToken newToken) =
            captable.createClass("Common", "TST-A", maxAuthorized, NO_CONVERSION_FLAG,
votingWeight, ALLOW_ALL_BOUNCER);
        captable.setManager(classId2, ISSUER, true);
        vm.stopPrank();

        vm.prank(ISSUER);
        captable.issue(NEW_USER, classId2, maxAuthorized);


        vm.prank(NEW_USER);
        newToken.delegate(NEW_USER);

        vm.roll(3);
        voting.quorum(2); // This reverts if there is an overflow
    }
```

Recommendation

One way to prevent this from happening is by maintaining the invariant that `votingWeight * abi.decode(returnData, (uint256))` does not overflow an `uint256`. A reasonable way to achieve this is by limiting authorized and voting weight. When creating a new class via `createClass()`, call `_weightedSumAllClasses()` and make sure that it does not overflow. If it does, revert the whole operation.

A similar strategy needs to be coded into `setAuthorized()` for all classes with voting weight greater than zero.

Status

Fixed on commit `06b077df73f2cb9b76ff2a6866a4498c716ce21d`.

The size of the `votingWeight`, `authorized` and `convertible` variables for each class were reduced to `uint16`, `uint128`, `uint128` respectively, preventing the scenario mentioned in this issue.

FRM-04 User transactions can be replayed after failure

Total Risk	Impact	Location
Medium	High	Relayer.sol
Fixed ✔	Likelihood Medium	

Description

Users are at risk of having their transactions replayed when using the relayer.

There are two reasons this might happen:

1. The user sends a bad nonce, higher than the one they should have sent.
2. The user's transactions or assertions fail.

In the first scenario, the signature with the bad nonce will get recorded on the blockchain. If the user continues to use the platform, at some point that nonce will become valid. At that time, an attacker can replay the user's transaction.

In the second scenario, the nonce is not increased because the whole transaction reverts. If the revert reason is temporary (eg: user has not enough funds) and the user does not send another transaction to invalidate the nonce, an attacker can replay the transaction when the condition that caused the original revert is not present anymore (eg: the user now has enough funds).

Test

```
    function testNonceIncreaseOnFailure() public {
        // We test here using an assertion failure, but any failure
        // besides nonce-mismatch (should invalidate all previous nonce)
        // and signature verification (should not change nonce)
        // should increase nonces by exactly one
        bytes32 actualReturnValue = bytes32(abi.encode(USER));
        bytes32 badExpectedValue = bytes32(uint256(0));

        FirmRelayer.Call memory call = _defaultCallWithData(address(target),
abi.encodeCall(target.onlySender, (USER)));
        FirmRelayer.Assertion memory assertion = FirmRelayer.Assertion(0, badExpectedValue);
        FirmRelayer.RelayRequest memory request = _defaultRequestWithCallAndAssertion(call,
assertion);
```

```
        uint256 previousNonce = relayer.getNonce(request.from);

        bytes32 hash = relayer.requestTypedDataHash(request);
        vm.expectRevert(
            abi.encodeWithSelector(FirmRelayer.AssertionFailed.selector, 0, actualReturnValue,
badExpectedValue)
        );
        relayer.relay(request, _signPacked(hash, USER_PK));

        uint256 afterNonce = relayer.getNonce(request.from);
        assertEq(previousNonce+1, afterNonce);
    }

    function testRelayNonceInvalidationOnBadNonce(uint256 badNonce) public {
        vm.assume(badNonce > 1);
        testBasicRelay();

        FirmRelayer.Call memory call = _defaultCallWithData(address(target),
abi.encodeCall(target.onlySender, (USER)));
        FirmRelayer.RelayRequest memory request = _defaultRequestWithCall(call);
        request.nonce = badNonce;

        bytes memory signature = _signPacked(relayer.requestTypedDataHash(request), USER_PK);
        vm.expectRevert(abi.encodeWithSelector(FirmRelayer.BadNonce.selector, uint256(1)));
        relayer.relay(request, signature);

        assertEq(target.lastSender(), USER);
        assertEq(relayer.getNonce(USER), badNonce+1);
    }
```

Recommendation

When a nonce higher than the one expected is sent, increase the next expected nonce by one.

When a transaction from the relayer reverts, make sure to increase the nonce anyway.

Status

The issue was correctly fixed. It is worth mentioning that Firm decided to not invalidate previous nonces if an incorrect nonce is sent. This mimics how Ethereum transactions behave. Clients should be aware of this and make sure that they send only correct nonces.

FRM-05	Managers are able to lock zero-balance account with vesting controller	
Total Risk Medium	Impact Medium	Location `src/captable/controllers/VestingController.sol`
Fixed ✔	Likelihood Medium	

Description

If an account with a balance of zero is assigned the vesting controller, the account becomes unusable. Any tokens sent to that account after the fact is effectively burnt.

This is because the vesting controller uses the amount to decide whether an account exists or not.

```
        if (account.amount == 0) {
            revert AccountDoesntExist();
        }
```

Because it deems the account non-existent, it will disallow any transfer of tokens from it. The user is not able either to `_cleanup` their controller, due to the same error.

Recommendation

Allow zero-amount accounts to call `cleanup` from the controller.

Status

Fixed on commit `2a3aea90e7a18581963bf5c75e43c25953410c5e`.

When setting a new controller in `Captable`, it is now checked that the balance of the controlled account is non zero. Coinspect identified that this change fixes the issue but restricts future Controller's implementations to use zero balance accounts. A less restrictive change would have been to modify this behavior directly in the `Vesting` controller.

FRM-06 Admin is given to unexpected users

Total Risk	Impact	Location
Medium	High	`src/budget/Budget.sol`
Fixed ✔	Likelihood Low	

Description

An admin clearance can be given to users unsuspectingly by a manager due to the way the admin bitmap is designed. Although the mechanics are documented by Firm, this behavior is so unexpected by end users that it will surely end up triggering in some real-life scenarios.

When creating a role, the `RoleManager` specifies an arbitrary admin mask. This mask is used to determine whether a user is an admin of a role or not. If the mask and the user roles share at least one bit, the user is an admin of that role.

The problem is that the mask can specify bits of roles that are not yet created. This leads to a confusing situation where:

1. Role A is created.
2. Role B is created.
3. User 1 is given membership to role A.
4. User 1 is given admin over Role B.
5. A new role C is created.
6. User 1 is given membership to role C.
7. Now the user is suddenly an admin of role A.

Test

```
    function testShouldNotCreateRolesWithMaskHigherThanExistingRoles(uint8 roles) public {
        // To avoid a situation like the one reproduced in testCanStepOverAdmin
        // a system needs to be in place where the admin mask cannot be set
        // to roles that do not yet exist.
```

```
        // This translates as avoiding setting masks above than 2**{amount of roles+1}
        for (uint8 i=0; i<roles; i++) {
            roles.createRole(bytes32(uint256(0x01)), "");
        }

        uint256 mask = 1 << (roles+1);
        vm.expectRevert();
        roles.createRole(bytes32(mask), "");
    }

    function testCanStepOverAdmin() public {
        vm.startPrank(address(safe));

        // roleCoinspect { id: 0x02, roleAdmins: 1111 0000, roleFlag: 0000 0010 }
        // (!) The issue lies here: roleCoinspect is setting as admin roles
        // that do not yet exist.
        uint8 roleCoinspect = roles.createRole(bytes32(uint256(0xF0)), "");

        // roleFirm { id: 0x03, roleAdmins: 0000 0100, roleFlag: 0000 0100}
        uint8 roleFirm = roles.createRole(bytes32(uint256(0x04)), "");

        assertEq(roleCoinspect, 0x02);
        assertEq(roleFirm, 0x03);

        // Joaquin is supposed to be an user of Firm, but not an admin of Coinspect
        address JOAQUIN = account("JOAQUIN");
        roles.setRole(JOAQUIN, roleCoinspect, true);

        // By giving Joaquin role of Coinspect, we set
        // their role to 0100 = 0x04, exactly the admin mask over firm
        assertFalse(roles.isRoleAdmin(JOAQUIN, roleCoinspect));
        assertTrue(roles.isRoleAdmin(JOAQUIN, roleFirm));

        // Some time passes, new roles are created.
        // Joaquin should be a part of these new roles.

        // role3 { id: 0x04, roleAdmins: 0000 0001, roleFlag: 0000 1000};
        uint8 role3 = roles.createRole(bytes32(uint256(0x01)), "");
        // role4 { id: 0x05, roleAdmins: 0000 0001, roleFlag: 0001 0000};
        uint8 role4 = roles.createRole(bytes32(uint256(0x01)), "");

        // By giving Joaquin these roles, their role bitmap is now
        // 0001 1100. The first nibble shares a bit with the admin
        // of coinspect.
        roles.setRole(JOAQUIN, role3, true);
        roles.setRole(JOAQUIN, role4, true);

        // Joaquin is now suddenly a role admin of coinspect.
        assertFalse(roles.isRoleAdmin(JOAQUIN, role3));
        assertFalse(roles.isRoleAdmin(JOAQUIN, role4));
        assertTrue(roles.isRoleAdmin(JOAQUIN, roleCoinspect));
    }
```

Recommendation

Do not allow to set an admin mask that involves bits of roles that do not exist yet. Alternatively, consider moving to a clearer structure with a mapping that establishes which users are admins of each role.

Status

The issue has been fixed. Roles can only be created with admins that already exist.

FRM-07 Fee on transfer tokens grant an excess of allowance

Total Risk	Impact	Location
Medium	Medium	`src/budget/Budget.sol`
Fixed ✔	Likelihood Medium	

Description

The `debitAllowance()` method on the Budget module does not consider tokens that have a fee on transfer.

This leads to the update on the allowance chain to be fed wrong data, as it will consider the **full amount** passed as the received by the vault, when this might not be the case.

```
        if (allowance.token != NATIVE_ASSET) {
            if (msg.value != 0) {
                revert NativeValueMismatch();
            }

            IERC20(allowance.token).safeTransferFrom(actor, address(safe()), amount);
        } else {
            if (msg.value != amount) {
                revert NativeValueMismatch();
            }

            payable(address(safe())).transfer(amount);
        }

        (nextResetTime,) = _checkAndUpdateAllowanceChain(allowanceId, amount, zeroCappedSub);
```

Even though the token is trusted, there are several bening and commonly used tokens which have fee on transfer mechanisms. Even more, some extremely common stablecoins like USDT support fees on transfer, they just have the fee rate set to zero. This might change without warning.

Test

The `fotToken` is a mock Fee-On-Transfer token whose implementation is in the appendix of this report.

```
  contract FeeOnTransferTokenTest is BudgetTest {
   ERC20FeeOnTransfer fotToken;
   function setUp() public override {
       super.setUp();


       fotToken = new ERC20FeeOnTransfer("FeeOnTr", "FOT", 18);
       fotToken.mint(address(safe), 100 ether);
       token = address(fotToken);


       vm.deal(address(safe), 100 ether);
   }


   function test_FeeOnTransfer() public {
       fotToken.setMaxFee(10 ether); // 10e18 tokens at most
       fotToken.setFeesBps(1000); // 10% fees
       uint256 mintAmount = 1 ether;
       fotToken.mint(SPENDER, mintAmount);


       vm.prank(SPENDER);
       fotToken.transfer(address(0x12345), mintAmount);


       assertTrue(fotToken.balanceOf(SPENDER) == (0));
       assertTrue(fotToken.balanceOf(address(0x12345)) == (mintAmount - mintAmount*1000 /
10000));
   }


   function test_ShouldNotGrantMoreAllowanceWithFeeOnTransfer() public {
       vm.prank(address(safe));
       uint256 expectedId = 1;


       uint256 allowanceId = budget.createAllowance(
           NO_PARENT_ID, SPENDER, address(token), 100, TimeShift(TimeShiftLib.TimeUnit.Daily,
0).encode(), ""
       );
       assertEq(allowanceId, expectedId);


       fotToken.setMaxFee(10 ether); // 10e18 tokens at most
       fotToken.setFeesBps(1000); // 10% fees
       uint256 initialSafeAmount = fotToken.balanceOf(address(safe));


       uint256 spent;
       vm.prank(SPENDER);
       budget.executePayment(allowanceId, RECEIVER, 100, "");
       (,, spent,,,,) = budget.allowances(allowanceId);
       assertEq(spent, 100);


       performDebit(RECEIVER, allowanceId, 50);
       (,, spent,,,,) = budget.allowances(allowanceId);
       assertLt(fotToken.balanceOf(address(safe)), initialSafeAmount - 50);
       assertGt(spent, 50);
   }
}
```

Recommendation

Use the difference between the safe's balance after and before instead of using the `amount` input parameter.

Status

Fixed on commit `321e0a649a5aa74368fd22bd67dc8c35b26e0a47`.

The `debitAllowance()` function now calculates the balances of the Safe before and after the `ERC20 transfer` and debits only its difference. Coinspect identified that the tests provided don't evaluate the usage of fee on transfer `ERC20's` and suggests adding that case to the suite.

FRM-08 Attackers are able to waste relayer's money

Total Risk	Impact	Location
Medium	Medium	`src/metatx/Relayer.sol` `src/budget/Budget.sol`
Fixed ✔	Likelihood Medium	

Description

An attacker can abuse the fact that the relayer will call arbitrary contracts to make Firm or any other relayer operator waste resources on gas.

The attacker needs to instruct the relayer to send a call to a contract that they control and which will return a lot of data, making the caller pay for the memory expansion cost. This is known as a `return bomb`.

Because the memory expansion is done on the context of the caller and not the callee, setting a max gas for the call will not prevent this attack.

```
    bytes memory payload = abi.encodePacked(call.data, asSender);
    (bool success, bytes memory returnData) = call.to.call{value: call.value, gas:
call.gas}(payload);
    if (!success) {
       revert CallExecutionFailed(i, call.to, returnData);
    }
```

The `Budget` module has a similar problem in the `executeMultiPayment` method, although the risk there is lower as the executor would have to be tricked into executing a payment to a malicious contract.

Recommendation

Use Nomad's Excessively Safe Call library to perform calls to untrusted contracts.

Status

Fixed by limiting how much return data is copied to the memory.

FRM-09 Safe can be taken over through votes

Total Risk	Impact	Location
Medium	High	`Voting.sol`
Fixed	Likelihood	
!	Low	

Description

If the Safe does not censor voting proposals (as described in FRM-02), then voters are too powerful. Voters can decide at will to execute arbitrary actions in the name of the Safe, including but not limited to draining it or updating modules with arbitrary implementations.

As the actions that the voters can execute are arbitrary, they are not easily reviewed. Voters might be tricked into voting for malicious proposals. In other scenarios, a token with voting power might be tradable and be available to outsiders in exchanges.

While it is expected that voters are able to remove or add members from the safe; it is recommended to *only* let them do that, and perform any other desired action once they have control of the safe through their address. This makes it easier to audit voting proposals by other voters.

The likelihood of the issue is considered low, as it has several preconditions depending on the exact vector. At the very least the Safe must have created a voting token, the voters must be malicious or tricked and the Safe must not censor the proposal. Nevertheless, the consequences would be **devastating** for a company.

Test

The following test shows how a proposal is able to drain the safe completely. In the appendix another example is provided where a module can be upgraded to an arbitrary implementation via the same mechanism.

```
function test_VaultDrain() public {
    // Issue EquityTokens to the first user
    address NEW_USER = address(0x123456);
    uint256 amount = 100;
```

```
    vm.prank(ISSUER);
    captable.issue(NEW_USER, classId, amount);

    // The user begins the attack by delegating those votes to themselves
    vm.prank(NEW_USER);
    token.delegate(NEW_USER);
    vm.roll(3);
    assertTrue(captable.getPastVotes(NEW_USER, 2) == 100);


    bytes memory createAllowanceCalldata = abi.encodeCall(Budget.createAllowance, (
        NO_PARENT_ID,
        NEW_USER,
        NATIVE_ASSET,
        address(safe).balance,
        TimeShift(TimeShiftLib.TimeUnit.Daily, 0).encode(),
        ""
    ));


    vm.prank(NEW_USER);
    uint256 proposalId = voting.propose(
        arr(address(budget)),
        arr(uint256(0)),
        arr(createAllowanceCalldata),
        "Safe Pwn");


    blocktravel(VOTING_DELAY + 1);

    vm.prank(NEW_USER);
    voting.castVote(proposalId, 1); // vote for
    blocktravel(VOTING_PERIOD);


    voting.execute(
        arr(address(budget)),
        arr(uint256(0)),
        arr(createAllowanceCalldata),
        keccak256(bytes("Safe Pwn"))
    );


    assertTrue(address(safe).balance == 1000 ether);
    assertTrue(address(NEW_USER).balance == 0 ether);


    // Get the current allowanceId of our recently created Allowance
    uint256 allowanceId = budget.allowancesCount() - 1;


    vm.prank(NEW_USER);
    budget.executePayment(1, NEW_USER, address(safe).balance, "");


    assertTrue(address(safe).balance == 0 ether);
    assertTrue(address(NEW_USER).balance == 1000 ether);
}
```

Recommendation

Do not allow the vault to be arbitrarily controlled by voting proposals. Instead, create an `onlyGovernance` modifier to explicitly state which actions can be executed via voting. Voting should, at least by default, only allow for removing and adding members of the safe.

If an organization intends to accept the risk of executing votes through a `delegatecall` from the safe, that option could be given as opt-in.

As a mitigation for accumulating voting power, consider changing the way votes are distributed. Tokens with voting power could have bouncers to disallow transfer by default (reproducing the behavior of an untradable soulbound token).

Status

Fixed with Pull request 184 which introduces the Semaphore on commit hash e28db7003629e01fc0179ef919c987365858aff0. See the notes on the Fixes review section of the audit.

FRM-10	Safe accepts money that it does not own		
Total Risk **Low**	Impact Medium	Location `src/budget/Budget.sol`	
Fixed !	Likelihood Low		

Description

The `debitAllowance()` method sends the whole **amount** to the Safe, but will cap the amount credited to the allowance id. This makes it possible to give money to the Vault by wrongly sending a higher amount than what the allowance spent.

Recommendation

Receive at most what the allowance id owes. Alternatively, allow users to explicitly specify that they are returning more funds than the allowed amount by checking a boolean input.

Status

Firm has stated that this is intended behavior to provide greater flexibility to the Budget module and considers this issue a **Will Not Fix**.

FRM-11 Multipayment recipient prevents others from being paid

Total Risk	Impact	Location
Low	Low	`src/budget/Budget.sol`
Fixed !	Likelihood Low	

Description

Because all the calls made on `__safeContext_performMultiTransfer` are required to succeed, a single failing call will make all the others fail as well.

This is generally not desired, and might impact payments that can otherwise go through.

```
        if (token == NATIVE_ASSET) {
            for (uint256 i = 0; i < length;) {
                (bool callSuccess,) = tos[i].call{value: amounts[i]}(hex"");
                require(callSuccess);
                unchecked {
                    i++;
                }
            }
        } else {
            for (uint256 i = 0; i < length;) {
                (bool callSuccess, bytes memory retData) =
                    token.call(abi.encodeCall(IERC20.transfer, (tos[i], amounts[i])));
                require(callSuccess && (((retData.length == 32 && abi.decode(retData, (bool)))
|| retData.length == 0)));
                unchecked {
                    i++;
                }
            }
        }
```

Recommendation

Continue processing payments even though one might fail. Let the caller know about failures by returning the indexes that have failed or emitting an event.

Status

Firm stated that this error can be caught in higher levels when simulating the transaction before sending it to the blockchain, and as such it is **Will Not Fix**.

FRM-12 Setting new allowance amount enables front running

Total Risk	Impact	Location
Low	Medium	`src/budget/Budget.sol`
Fixed	Likelihood	
!	Low	

Description

The spender of an `allowanceId` has an incentive to spend the allowed amount by frontrunning a `setAllowanceAmount()` call.

The admin of an allowance can set a new allowance amount. This process could be abused by the spender via a frontrunning, as the allowance spender will always want to get the most tokens from the vault. In the event of detecting an allowance decrease, the spender has incentives to execute a payment to use up all his allowed amount:

- Alice has an allowance with the `ID = 10` to spend 100 tokens A.
- The allowance admin wants to reduce her allowance and calls `setAllowanceAmount(10, 50)`.
- Alice frontruns that call and spends 100 tokens by executing a payment to an arbitrary account managed by herself.
- When the admin call is mined, Alice's allowance is 50 and her `allowance.spent = 100`.

This scenario opens a race condition where every spender has incentives to frontrun an allowance decrease call.

Recommendation

Add a parameter to the `setAllowance` method, `expectedSpent`. A call to `setAllowance(user, newAllowance, expectedSpent)` should result in the user having an allowance of `newAllowance - (user.spent - expectedSpent))`.

This results in the user being able to use at most their previous allowance and no more.

Alternatively, Implement a `commit-reveal` process to set new allowance amounts.

Status

Firm stated that this issue is not a concern in their threat model, and as such it is **Will Not Fix**.

FRM-13 Captable accepts 129 classes

Total Risk	Impact	Location
Info	-	`src/captable/Captable.sol`
Fixed ✔	**Likelihood** -	

Description

Due to an off-by-one error in the initialization of the Captable.sol contract, the maximum number of classes is 129; not 128.

```
        unchecked {
            if ((classId = classCount++) >= CLASSES_LIMIT) {
                revert ClassCreationAboveLimit();
            }
        }
```

The code above allows for the creation of classes 0 to 128, resulting in 129 total classes.

Even though there is no real impact in the contract except for a slightly higher max gas usage than expected; frontends and other systems might rely on the cap.

Recommendation

Fix the off-by-one error: `classId` should be at most 127. Alternatively, modify the documentation saying that the maximum number of classes is 129.

Status

Firm has stated that this is expected behavior. There is no risk as long as the max amount of classes is correctly documented and frontends take this into account.

5. Appendix

1. Fee On Transfer Mock Token Implementation

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.17;


import "openzeppelin/access/Ownable.sol";
import "openzeppelin/token/ERC20/ERC20.sol";
import "openzeppelin/utils/Context.sol";
import "openzeppelin/token/ERC20/IERC20.sol";


contract ERC20Fees is Context, IERC20{
   mapping(address => uint256) private _balances;


   mapping(address => mapping(address => uint256)) private _allowances;


   uint256 private _totalSupply;


   string private _name;
   string private _symbol;


   uint8 private _decimals;


   uint256 private _feeRateBPS = 0;
   uint256 private _maxFeeAmount = 0;


   address public feeRecipient;


   constructor(string memory name_, string memory symbol_, uint8 decimals_) {
       _name = name_;
       _symbol = symbol_;
       _decimals = decimals_;
       feeRecipient = msg.sender;
   }


   function decimals() public view virtual returns (uint8) {
       return _decimals;
   }
   function feeRate() public view virtual returns (uint256) {
       return _feeRateBPS;
   }
   function maxFee() public view virtual returns (uint256) {
       return _maxFeeAmount;
   }


   function _setFeeBps(uint256 _newFeeBPS) internal {
       _feeRateBPS =_newFeeBPS;
   }


   function _setMaxFee(uint256 _newMax) internal {
       _maxFeeAmount =_newMax;
   }

   /**
    * @dev Returns the name of the token.
```

```
 */
function name() public view virtual returns (string memory) {
    return _name;
}


/**
 * @dev Returns the symbol of the token, usually a shorter version of the
 * name.
 */
function symbol() public view virtual returns (string memory) {
    return _symbol;
}


 /**
 * @dev See {IERC20-totalSupply}.
 */
function totalSupply() public view virtual override returns (uint256) {
    return _totalSupply;
}


/**
 * @dev See {IERC20-balanceOf}.
 */
function balanceOf(address account) public view virtual override returns (uint256) {
    return _balances[account];
}


/**
 * @dev See {IERC20-transfer}.
 *
 * Requirements:
 *
 * - `to` cannot be the zero address.
 * - the caller must have a balance of at least `amount`.
 */
function transfer(address to, uint256 amount) public virtual override returns (bool) {
    address owner = _msgSender();
    _transfer(owner, to, amount);
    return true;
}


/**
 * @dev See {IERC20-allowance}.
 */
function allowance(address owner, address spender) public view virtual override returns (uint256) {
    return _allowances[owner][spender];
}


/**
 * @dev See {IERC20-approve}.
 *
 * NOTE: If `amount` is the maximum `uint256`, the allowance is not updated on
 * `transferFrom`. This is semantically equivalent to an infinite approval.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
function approve(address spender, uint256 amount) public virtual override returns (bool) {
    address owner = _msgSender();
    _approve(owner, spender, amount);
    return true;
}


/**
 * @dev See {IERC20-transferFrom}.
```

```
     *
     * Emits an {Approval} event indicating the updated allowance. This is not
     * required by the EIP. See the note at the beginning of {ERC20}.
     *
     * NOTE: Does not update the allowance if the current allowance
     * is the maximum `uint256`.
     *
     * Requirements:
     *
     * - `from` and `to` cannot be the zero address.
     * - `from` must have a balance of at least `amount`.
     * - the caller must have allowance for ``from``'s tokens of at least
     * `amount`.
     */
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        address spender = _msgSender();
        _spendAllowance(from, spender, amount);
        _transfer(from, to, amount);
        return true;
    }


    /**
     * @dev Atomically increases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, allowance(owner, spender) + addedValue);
        return true;
    }


    /**
     * @dev Atomically decreases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
{
        address owner = _msgSender();
        uint256 currentAllowance = allowance(owner, spender);
        require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
        unchecked {
            _approve(owner, spender, currentAllowance - subtractedValue);
        }


        return true;
    }
```

```
   /**
    * @dev Moves `amount` of tokens from `from` to `to`.
    *
    * This internal function is equivalent to {transfer}, and can be used to
    * e.g. implement automatic token fees, slashing mechanisms, etc.
    *
    * Emits a {Transfer} event.
    *
    * Requirements:
    *
    * - `from` cannot be the zero address.
    * - `to` cannot be the zero address.
    * - `from` must have a balance of at least `amount`.
    */
   function _transfer(
       address from,
       address to,
       uint256 amount
   ) internal virtual {
       require(from != address(0), "ERC20: transfer from the zero address");
       require(to != address(0), "ERC20: transfer to the zero address");

       uint256 fromBalance = _balances[from];
       require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");

       // This returns zero if the divisor is greater than the numerator, only for dust.
       uint256 fee = (amount * _feeRateBPS) / 10000;
       if (fee > _maxFeeAmount) {
           fee = _maxFeeAmount;
       }
       uint256 sendAmount = amount - fee;
       unchecked {
           _balances[from] = fromBalance - amount;
           _balances[to] += sendAmount;
       }


       if (fee > 0) {
           _balances[feeRecipient] = _balances[feeRecipient] + fee;
           emit Transfer(msg.sender, feeRecipient, fee);
       }


       emit Transfer(from, to, amount);
   }


   /** @dev Creates `amount` tokens and assigns them to `account`, increasing
    * the total supply.
    *
    * Emits a {Transfer} event with `from` set to the zero address.
    *
    * Requirements:
    *
    * - `account` cannot be the zero address.
    */
   function _mint(address account, uint256 amount) internal virtual {
       require(account != address(0), "ERC20: mint to the zero address");


       _totalSupply += amount;
       unchecked {
           // Overflow not possible: balance + amount is at most totalSupply + amount, which is
checked above.
           _balances[account] += amount;
       }
       emit Transfer(address(0), account, amount);
   }


   /**
    * @dev Destroys `amount` tokens from `account`, reducing the
    * total supply.
```

```
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * Requirements:
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");


        uint256 accountBalance = _balances[account];
        require(accountBalance >= amount, "ERC20: burn amount exceeds balance");
        unchecked {
            _balances[account] = accountBalance - amount;
            // Overflow not possible: amount <= accountBalance <= totalSupply.
            _totalSupply -= amount;
        }


        emit Transfer(account, address(0), amount);


    }


    /**
     * @dev Sets `amount` as the allowance of `spender` over the `owner` s tokens.
     *
     * This internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");


        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }


    /**
     * @dev Updates `owner` s allowance for `spender` based on spent `amount`.
     *
     * Does not update the allowance amount in case of infinite allowance.
     * Revert if not enough allowance is available.
     *
     * Might emit an {Approval} event.
     */
    function _spendAllowance(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
        uint256 currentAllowance = allowance(owner, spender);
        if (currentAllowance != type(uint256).max) {
            require(currentAllowance >= amount, "ERC20: insufficient allowance");
            unchecked {
                _approve(owner, spender, currentAllowance - amount);
            }
```

```
        }
    }
}


contract ERC20FeeOnTransfer is ERC20Fees, Ownable {


    constructor(string memory name_, string memory symbol_, uint8 decimals_) ERC20Fees(name_, symbol_,
decimals_) {
    }


    function mint(address to, uint256 amount) external onlyOwner {
        _mint(to, amount);
    }


    function setFeesBps(uint256 amount) external onlyOwner {
        _setFeeBps(amount);
    }
    function setMaxFee(uint256 amount) external onlyOwner {
        _setMaxFee(amount);
    }


}
```

2. Upgrade to arbitrary modules and drain the safe

```
function test_CannotUpgradeToArbitraryImpl() public {
      bytes32 IMPL_SLOT = 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;
      // Issue EquityTokens to the first user
      address NEW_USER = address(0x123456);
      uint256 amount = 100;


      vm.prank(ISSUER);
      captable.issue(NEW_USER, classId, amount);

      // The user begins the attack by delegating those votes to themselves
      vm.prank(NEW_USER);
      token.delegate(NEW_USER);
      vm.roll(3);
      assertTrue(captable.getPastVotes(NEW_USER, 2) == 100);


      FakeModule fakeModule = new FakeModule();


      bytes memory createAllowanceCalldata = abi.encodeCall(EIP1967Upgradeable.upgrade, (
          IModuleMetadata(address(fakeModule))
      ));


      vm.prank(NEW_USER);
      uint256 proposalId = voting.propose(
          arr(address(captable)),
          arr(uint256(0)),
          arr(createAllowanceCalldata),
          "Safe Pwn");
```

```
      blocktravel(VOTING_DELAY + 1);

      vm.prank(NEW_USER);
      voting.castVote(proposalId, 1); // vote for
      blocktravel(VOTING_PERIOD);


      voting.execute(
          arr(address(captable)),
          arr(uint256(0)),
          arr(createAllowanceCalldata),
          keccak256(bytes("Safe Pwn"))
      );

      bytes32 implBytes32 = vm.load(address(captable), IMPL_SLOT);
      assertEq(implBytes32, bytes32(uint256(uint160(address(fakeModule)))));
  }
```

```
contract FakeModule is IModuleMetadata {
  function moduleId() external pure returns (string memory){
      return "pwn";
  }


  function moduleVersion() external pure returns (uint256){
      return 0;
  }
}



  function test_VaultDrain() public {
      // Issue EquityTokens to the first user
      address NEW_USER = address(0x123456);
      uint256 amount = 100;


      vm.prank(ISSUER);
      captable.issue(NEW_USER, classId, amount);

      // The user begins the attack by delegating those votes to themselves
      vm.prank(NEW_USER);
      token.delegate(NEW_USER);
      vm.roll(3);
      assertTrue(captable.getPastVotes(NEW_USER, 2) == 100);


      bytes memory createAllowanceCalldata = abi.encodeCall(Budget.createAllowance, (
          NO_PARENT_ID,
          NEW_USER,
          NATIVE_ASSET,
          address(safe).balance,
          TimeShift(TimeShiftLib.TimeUnit.Daily, 0).encode(),
          ""
      ));


      vm.prank(NEW_USER);
      uint256 proposalId = voting.propose(
          arr(address(budget)),
          arr(uint256(0)),
          arr(createAllowanceCalldata),
          "Safe Pwn");


      blocktravel(VOTING_DELAY + 1);

      vm.prank(NEW_USER);
      voting.castVote(proposalId, 1); // vote for
      blocktravel(VOTING_PERIOD);
```

```
    voting.execute(
        arr(address(budget)),
        arr(uint256(0)),
        arr(createAllowanceCalldata),
        keccak256(bytes("Safe Pwn"))
    );


    assertTrue(address(safe).balance == 1000 ether);
    assertTrue(address(NEW_USER).balance == 0 ether);


    // Get the current allowanceId of our recently created Allowance
    uint256 allowanceId = budget.allowancesCount() - 1;


    vm.prank(NEW_USER);
    budget.executePayment(1, NEW_USER, address(safe).balance, "");


    assertTrue(address(safe).balance == 0 ether);
    assertTrue(address(NEW_USER).balance == 1000 ether);
}
```

6. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any off-chain systems or frontends that communicate with the contracts, nor the general operational security of the organization that developed the code.