Rigor Protocol contest Findings & Analysis Report

2022-09-12

Table of contents

- [M-01] `Project.changeOrder()` would work unexpectedly for non SCConfirmed tasks.
- [M-02] Missing upper limit definition in `replaceLenderFee()` of `HomeFi.sol`
- [M-03] Signature Checks could be passed when `SignatureDecoder.recoverKey()` returns 0
- [M-04] Hash approval not possible when contractor == subcontractor
- [M-05] Anyone can create disputes if `contractor` is not set
- [M-06] Attacker can drain all the projects within minutes, if admin account has been exposed
- [M-07] `Project.raiseDispute()` doesn't use approvedHashes - meaning users who use contracts can't raise disputes
- [M-08] Builders must pay more interest when the system is paused.
- [M-09] It should not submit a project with no total budget. Requires at least one task with cost > 0
- [M-10] Possible DOS in `lendToProject()` and `toggleLendingNeeded()` function because unbounded loop can run out of gas
- [M-11] Owner of project NFT has no purpose
- [M-12] `updateProjectHash` does not check project address
- [M-13] In `Project.setComplete()` , the signature can be reused when the first call is reverted for some reason
- [M-14] Incorrect initialization of smart contracts with Access Control issue
- [M-15] `Project.addTasks()` wouldn't work properly when it's called from disputes contract.
- [M-16] New subcontractor can be set for a SCConfirmed task without current subcontractor consent
- [M-17] Malicious delegated contractor can block funding tasks or mark tasks as complete
- [M-18] Task Functionality completely sidestepped via `autoWithdraw`
- [M-19] `changeOrder` requires subcontractor signature when the subcontractor address is 0

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Rigor Protocol smart contract system written in Solidity. The audit contest took place between August 1—August 6 2022.

Wardens

139 Wardens contributed reports to the Rigor Protocol contest:

1. Lambda
2. hansfriese
3. rbserver
4. 0x52

5. 0xA5DF
6. indijanc
7. GalloDaSballo
8. wastewa
9. vlad_bochok
10. byndooa
11. 0x1f8b
12. hyh
13. berndartmueller
14. sseefried
15. cccz
16. cryptphi
17. minhquanym
18. horsefacts
19. scaraven
20. Haipls
21. TrungOre
22. defsec
23. Deivitto
24. 0xNazgul
25. MEP
26. Chom
27. Bahurum
28. bin2chen
29. cryptonue
30. c3phas
31. arcoun
32. rotcivegaf
33. simon135

34. obront
35. saneryee
36. rokinot
37. ElKu
38. auditor0517
39. smiling_heretic
40. kankodu
41. kaden
42. aez121
43. neumo
44. Dravee
45. Ruhum
46. __141345__
47. 8olidity
48. panprog
49. codexploder
50. lllllll
51. CertoraInc (egjlmn1, OriDabush, ItayG, and shakedwinder)
52. fatherOfBlocks
53. saian
54. Guardian
55. MiloTruck
56. JC
57. benbaessler
58. hake
59. 0xkatana
60. gogo
61. oyc_109
62. pfapostol

63. samruna
64. joestakey
65. ReyAdmirado
66. TomJ
67. Rolezn
68. brgltd
69. Extropy
70. Aymen0909
71. 0xSmartContract
72. delfin454000
73. mics
74. bobirichman
75. sikorico
76. ak1
77. _Adam
78. robee
79. SooYa
80. CodingNameKiki
81. erictee
82. Bnke0x0
83. ajtra
84. Tomio
85. Funen
86. Rohan16
87. Sm4rty
88. Waze
89. sach1r0
90. ignacio
91. supernova

92. asutorufos
93. 0xSolus
94. Noah3o6
95. a12jmx
96. djxploit
97. Ch_301
98. dipp
99. 0xf15ers (remora and twojoy)
100. bulej93
101. Soosh
102. 0xNineDec
103. exd0tpy
104. ayeslick
105. poirots (DavideSilva, resende, and naps62)
106. Yiko
107. 0xsolstars (Varun_Verma and masterchief)
108. p_crypt0
109. Jujic
110. Throne6g
111. NoamYakov
112. Chinmay
113. eierina
114. jag
115. bharg4v
116. 0x040
117. ballx
118. durianSausage
119. Metatron
120. ret2basic

121. 0xsam
122. Fitraldys
123. gerdusx
124. kyteg
125. apostle0x01
126. lucacez
127. dharma09
128. PaludoX0
129. zeesaw
130. 0xc0ffEE
131. tofunmi
132. teddav

This contest was judged by Jack the Pug.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 26 unique vulnerabilities. Of these vulnerabilities, 6 received a risk rating in the category of HIGH severity and 20 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 99 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 91 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Rigor Protocol contest repository, and is composed of 7 smart contracts (and 2 libraries) written in the Solidity programming language and includes 1,724 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (6)

[H-01] Untyped data signing

Submitted by Lambda, also found by 0x1f8b, 0x52, horsefacts, vlad_bochok, and wastewa

Community.sol#L175
Community.sol#L213
Community.sol#L530
Disputes.sol#L91
Project.sol#L142
Project.sol#L167
Project.sol#L235
Project.sol#L286
Project.sol#L346
Project.sol#L402
Project.sol#L499

In many places of the project (see affected code), untyped application data is directly hashed and signed. This is strongly disencouraged, as it enables different attacks (that each could be considered their own issue / vulnerability, but I submitted it as one, as they have all the same root cause):

1.) Signature reuse across different Rigor projects:
While some signature contain the project address, not all do. For instance, `updateProjectHash` only contains a `_hash` and a `_nonce`. Therefore, we can have the following scenario: Bob is the owner of project A and signs / submit `updateProjectHash` with nonce 0 and some hash. Then, a project B that also has Bob as the owner is created. Attacker Charlie can simply take the `_data` and `_signature` that Bob previously submitted to project A and send it to project B. As this project will have a nonce of 0 (fresh created), it will accept it. `updateTaskHash` is also affected by this.
2.) Signature reuse across different chains:
Because the chain ID is not included in the data, all signatures are also valid when the project is launched on a chain with another chain ID. For instance, let's say it is also launched on Polygon. An attacker can now use all of the Ethereum signatures there. Because the Polygon addresses of user's (and potentially contracts, when the nonces for creating are the same) are often identical, there can be situations where the payload is meaningful on both chains.
3.) Signature reuse across Rigor functions:
Some functions accept and decode data / signatures that were intended for other functions. For instance, see this example of providing the data & signature that was intended for `inviteContractor` to `setComplete`:

```
diff --git a/test/utils/projectTests.ts b/test/utils/projectTest
index ae9e202..752e01f 100644
--- a/test/utils/projectTests.ts
+++ b/test/utils/projectTests.ts
@@ -441,7 +441,7 @@ export const projectTests = async ({
       }
     });

-   it('should be able to invite contractor', async () => {
+   it.only('should be able to invite contractor', async () => {
      expect(await project.contractor()).to.equal(ethers.constant
      const data = {
        types: ['address', 'address'],
@@ -452,6 +452,7 @@ export const projectTests = async ({
        signers[1],
      ]);
      const tx = await project.inviteContractor(encodedData, sigr
+    const tx2 = await project.setComplete(encodedData, signatur
      await expect(tx)
        .to.emit(project, 'ContractorInvited')
        .withArgs(signers[1].address);
```

While this reverts because there is no task that corresponds to the address that is signed there, this is not always the case.
4.) Signature reuse from different Ethereum projects & phishing
Because the payload of these signatures is very generic (two addresses, a byte and two uints), there might be situations where a user has already signed data with the same format for a completely different Ethereum application. Furthermore, an attacker could set up a DApp that uses the same format and trick someone into signing the data. Even a very security-conscious owner that has audited the contract of this DApp (that does not have any vulnerabilities and is not malicious, it simply consumes signatures that happen to have the same format) might be willing to sign data for this DApp, as he does not anticipate that this puts his Rigor project in danger.

Recommended Mitigation Steps

I strongly recommend to follow EIP-712 and not implement your own standard / solution. While this also improves the user experience, this topic is very complex and not easy to get right, so it is recommended to use a battle-tested approach that people have thought in detail about. All of the mentioned attacks are not possible with EIP-712:
1.) There is always a domain separator that includes the contract address.
2.) The chain ID is included in the domain separator
3.) There is a type hash (of the function name / parameters)
4.) The domain separator does not allow reuse across different projects, phishing with an innocent DApp is no longer possible (it would be shown to the user that he is signing data for Rigor, which he would off course not do on a different site)

parv3213 (Rigor) confirmed

[H-02] Builder can halve the interest paid to a community owner due to arithmetic rounding

Submitted by scaraven, also found by 0x52, auditor0517, Deivitto, hansfriese, Lambda, rbserver, simon135, smiling_heretic, sseefried, and TrungOre

Community.sol#L685-L686

Due to arithmetic rounding in `returnToLender()`, a builder can halve the APR paid to a community owner by paying every 1.9999 days. This allows a builder to drastically decrease the amount of interest paid to a community owner, which in turn allows them to advertise very high APR rates to secure funding, most of which they will not pay.

This issue occurs in the calculation of `noOfDays` in `returnToLender()` which calculates the number of days since interest has last been calculated. If a builder repays a very small amount of tokens every 1.9999 days, then the `noOfDays` will be rounded down to `1 days` however `lastTimestamp` is updated to the current timestamp anyway, so the builder essentially accumulates only 1 day of interest after 2 days.

I believe this is high severity because a community owner can have a drastic decrease in interest gained from a loan which counts as lost rewards. Additionally, this problem does not require a malicious builder because if a builder pays at a wrong time, the loaner receives less interest anyway.

Proof of Concept

1. A community owner provides a loan of 500_000 tokens to a builder with an APR of 10% (ignoring treasury fees)
2. Therefore, the community owner will expect an interest of 136.9 tokens per day (273.9 per 2 days)
3. A builder repays 0.000001 tokens at `lastTimestamp + 2*86400 - 1`
4. `noOfDays` rounds down to 1 thereby accumulating `500_000 * 100 * 1 / 365000 = 136` tokens for 2 days
5. Therefore, the community owner only receives 5% APR with negligible expenses for the builder

Tools Used

VS Code

Recommended Mitigation Steps

There are two possible mitigations:

1. Add a scalar to `noOfDays` so that any rounding which occurs is negligible

i.e.

```
        uint256 _noOfDays = (block.timestamp -
            _communityProject.lastTimestamp) * SCALAR / 86400; /

        /// Interest formula = (principal * APR * days) / (365 *
        // prettier-ignore
        uint256 _unclaimedInterest =
                _lentAmount *
                _communities[_communityID].projectDetails[_proje
                _noOfDays /
                365000 /
                SCALAR;
```

2. Remove the `noOfDays` calculation and calculate interest in one equation which reduces arithmetic rounding

```
uint256 _unclaimedInterest =
                _lentAmount *
                _communities[_communityID].projectDetails[_proje
                (block.timestamp -
            _communityProject.lastTimestamp) /
                365000 /
                86400;
```

[zgorizzo69 (Rigor) confirmed]

[H-03] Builder can call `Community.escrow` again to reduce debt further using same signatures

Submitted by sseefried, also found by 0xA5DF, Bahurum, bin2chen, byndooa, cccz, GalloDaSballo, hyh, kankodu, Lambda, and minhquanym

[Community.sol#L509]

Since there is no nonce in the data decoded at the beginning of function `escrow`, a builder can call the function multiple times reducing their debt as much as they wish.

Proof of Concept

- A builder has a debt of $50,000
- A lender, a builder, and an escrow agent all ~~enter a bar~~ sign a message that will reduce the debt of the builder by $5,000, upon receipt of physical cash.
- Function `escrow` is called and debt is reduced to $45,000.
- The builder, using the same `_data` and `_signature` then calls `escrow` a further 9 times reducing their debt to zero.

Recommended Mitigation Steps

1. Similar to function `publishProject`, add a new field into the [ProjectDetails] struct called `escrowNonce`.
2. Modify function `escrow` to check this nonce and update it after the debt has been reduced.

See the diff below for full changes.

```
diff --git a/contracts/Community.sol b/contracts/Community.sol
index 1585670..b834d0e 100644
--- a/contracts/Community.sol
+++ b/contracts/Community.sol
@@ -15,7 +15,7 @@ import {SignatureDecoder} from "./libraries/Si

  /**
   * @title Community Contract for HomeFi v2.5.0
-
+
  * @notice Module for coordinating lending groups on HomeFi prc
  */
 contract Community is
@@ -520,10 +520,11 @@ contract Community is
             address _agent,
             address _project,
             uint256 _repayAmount,
+            uint256 _escrowNonce,
             bytes memory _details
         ) = abi.decode(
                 _data,
-                (uint256, address, address, address, address, u
+                (uint256, address, address, address, address, u
             );

         // Compute hash from bytes
@@ -540,6 +541,12 @@ contract Community is
             _lender == _communities[_communityID].owner,
             "Community::!Owner"
         );
+        ProjectDetails storage _communityProject =
+          _communities[_communityID].projectDetails[_project];
+        require(
+            _escrowNonce == _communityProject.escrowNonce,
+            "Community::invalid escrowNonce"
+        );

         // check signatures
         checkSignatureValidity(_lender, _hash, _signature, 0);
@@ -548,6 +555,7 @@ contract Community is

         // Internal call to reduce debt
         _reduceDebt(_communityID, _project, _repayAmount, _deta
+        _communityProject.escrowNonce = _communityProject.escrc
         emit DebtReducedByEscrow(_agent);
     }

diff --git a/contracts/interfaces/ICommunity.sol b/contracts/int
index c45bbf0..652f51c 100644
--- a/contracts/interfaces/ICommunity.sol
+++ b/contracts/interfaces/ICommunity.sol
@@ -29,6 +29,7 @@ interface ICommunity {
         uint256 lentAmount; // current principal lent to projec
         uint256 interest; // total accrued interest on `lentAmc
         uint256 lastTimestamp; // timestamp when last lending /
+        uint256 escrowNonce; // signing nonce to use when reduc
```

```
        }
```

zgorizzo69 (Rigor) confirmed

[H-04] Project funds can be drained by reusing signatures, in some cases

Submitted by 0xA5DF, also found by Bahurum, bin2chen, byndooa, cryptphi, hansfriese, horsefacts, kaden, Lambda, neumo, panprog, rokinot, scaraven, and sseefried

Project.sol#L386-L490
Project.sol#L330-L359
Tasks.sol#L153-L164

This attack path is the results of signatures reusing in 2 functions - `changeOrder()` and `setComplete()` , and a missing modifier at `Tasks.unApprove()` library function.

Impact

Draining the project from funds

Current or previous subcontractor of a task can drain the project out of its funds by running `setComplete()` multiple times.

This can be exploited in 3 scenarios:

- The price of a task was changed to a price higher than available funds (i.e. `totalLent - _totalAllocated` , and therefore gets unapproved), and than changed back to the original price (or any price that's not higher than available funds)
- The subcontractor for a task was changed via `changeOrder` and then changed back to the original subcontractor

    - e.g. - Bob was the original SC, it was changed to Alice, and then back to Bob
- Similar to the case above, but even if the current SC is different from the original SC - it can still work if the current and previous SCs are teaming up to run the attack
    - e.g. Bob was the original SC, it was changed to Alice, and changed again to Eve. And now Alice and Eve are teaming up to drain funds from the project

After `setComplete()` ran once by the legitimate users (i.e. signed by contractor, SC and builder), the attackers can now run it multiple times:

- Reuse signatures to run `changeOrder()` - changing SC or setting the price to higher than available funds
    - The only signer that might change is the subcontractor, he's either teaming up with the attacker (scenario #3) or he was the SC when it was first called (scenario #2)
- In case of price change:
    - change it back to the original price via `changeOrder()` , reusing signatures
    - Run `allocateFunds()` to mark it as funded again
- SC runs `acceptInvite()` to mark task as active
- Run `setComplete()` reusing signatures
    - If SC has changed - replace his signature with the current one (current SC should be one of the attackers)
- Repeat till the project runs out of funds

Changing tasks costs/subcontractor by external users

This can also be used by external users (you don't need to be builder/GC/SC in order to run `changeOrder()`) to troll the system (This still requires the task to be changed at least twice, otherwise re-running `changeOrder()` with the same data would have no effect).

- Changing the task cost up or down, getting the SC paid a different amount than intended (if it goes unnoticed, or front-run the `setComplete()` function)
- Unapproving a task by setting a different SC or a price higher than available funds
    - The legitimate users can change it back, but the attacker can change it again, both sides playing around till someone gets tired :)

Proof of Concept

Since the tests depend on each other, the PoC tests were created by adding them to the file `test/utils/projectTests.ts`, after the function `it('should be able to complete a task'` (Line 1143).

In the first test - a subcontractor is changed and then changed back.
In the second scenario a price is changed to the new price (that is higher than the total available funds, and therefore is unapproved) and then back to its original price (it can actually be any price that is not higher than the available funds).
In both cases I'm demonstrating how the project can be drained out of fund.

Note: see warden's original submission *for full proof of concept.*

Tools Used

Hardhat

Recommended Mitigation Steps

- Use nonce to protect `setComplete()` and `changeOrder()` from signatures reuse
- Add the `onlyActive()` modifier to `Tasks.unApprove()`
- Consider limiting `allocateFunds()` for builder only (this is not necessary to resolve the bug, just for hardening security)

zgorizzo69 (Rigor) confirmed and commented:

> Very nice wrap up.

[H-05] Add members to the not yet created community

Submitted by vlad_bochok, also found by indijanc, Lambda, and wastewa

[Community.sol#L187](#)
[Community.sol#L179](#)
[Community.sol#L878](#)
[SignatureDecoder.sol#L39](#)

There is a `addMember` function in the `Community`. The function accepts `_data` that should be signed by the `_community.owner` and `_newMemberAddr`.

```
        // Compute hash from bytes
        bytes32 _hash = keccak256(_data);

        // Decode params from _data
        (
            uint256 _communityID,
            address _newMemberAddr,
            bytes memory _messageHash
        ) = abi.decode(_data, (uint256, address, bytes));

        CommunityStruct storage _community = _communities[_commu

        // check signatures
        checkSignatureValidity(_community.owner, _hash, _signatu
        checkSignatureValidity(_newMemberAddr, _hash, _signature
```

The code above shows exactly what the contract logic looks like.

1. `_communityID` is taken from the data provided by user, so it can arbitrarily. Specifically, community with selected `_communityID` can be not yet created. For instance, it can be equal to the `communityCount + 1`, thus the next created community will have this `_communityID`.
2. `_communities[_communityID]` will store null values for all fields, for a selected `_communityID`. That means, `_community.owner == address(0)`
3. `checkSignatureValidity` with a parameters `address(0), _hash, _signature, 0` will not revert a call if an attacker provide incorrect

`_signature`.

Let's see the implementation of `checkSignatureValidity`:

```
            // Decode signer
            address _recoveredSignature = SignatureDecoder.recoverKe
                _hash,
                _signature,
                _signatureIndex
            );

            // Revert if decoded signer does not match expected addr
            // Or if hash is not approved by the expected address.
            require(
                _recoveredSignature == _address || approvedHashes[_a
                "Community::invalid signature"
            );

            // Delete from approvedHash. So that signature cannot be
            delete approvedHashes[_address][_hash];
```

No restrictions on `_recoveredSignature` or `_address`. Moreover, if `SignatureDecoder.recoverKey` can return zero value, then there will be no revert.

```
           if (messageSignatures.length % 65 != 0) {
                return (address(0));
            }

            uint8 v;
            bytes32 r;
            bytes32 s;
            (v, r, s) = signatureSplit(messageSignatures, pos);

            // If the version is correct return the signer address
            if (v != 27 && v != 28) {
                return (address(0));
            } else {
                // solium-disable-next-line arg-overflow
                return ecrecover(toEthSignedMessageHash(messageHash)
            }
```

As we can see below, `recoverKey` function can return zero value, if an `ecrecover` return zero value or if `v != 27 || v != 28`. Both cases are completely dependent on the input parameters to the function, namely from `signature` that is provided by attacker.

4. `checkSignatureValidity(_newMemberAddr, _hash, _signature, 1)` will not revert the call if an attacker provide correct signature in the function. It is obviously possible.

All in all, an attacker can add as many members as they want, BEFORE the `community` will be created.

Recommended Mitigation Steps

1. `checkSignatureValidity` / `recoverKey` should revert the call if an `address == 0`.
2. `addMember` should have a `require(_communityId <= communityCount)`

[parv3213 (Rigor) confirmed]

[Jack the Pug (judge) commented]:

> Nice catch!

> Btw, this `v != 27 && v != 28` check is no longer needed:

```
if (v != 27 && v != 28) {
            return (address(0));
}
```

> See: https://twitter.com/alexberegszaszi/status/1534461421454606336?s=20&t=H0Dv3ZT2bicx00hLWJk7Fg

[H-06] Wrong APR can be used when project is unpublished and published again

Submitted by Lambda, also found by rbserver

Community.sol#L267

When a project is unpublished from a community, it can still owe money to this community (on which it needs to pay interest according to the specified APR). However, when the project is later published again in this community, the APR can be overwritten and the overwritten APR is used for the calculation of the interest for the old project (when it was unpublished).

Proof Of Concept

1.) Project A is published in community I with an APR of 3%. The community lends 1,000,000 USD to the project.
2.) Project A is unpublished, the `lentAmount` is still 1,000,000 USD.
3.) During one year, no calls to `repayLender`, `reduceDebt`, or `escrow` happens, i.e. the interest is never added and the `lastTimestamp` not updated.
4.) After one year, the project is published again in the same community. Because the FED raised interest rates, it is specified that the APR should be 5% from now on.
5.) Another $1,000,000 is lent to the project by calling `lendToProject`. Now, `claimInterest` is called which calculates the interest of the last year for the first million. However, the function already uses the new APR of 5%, meaning the added interest is 50,000 USD instead of the correct 30,000 USD.

Recommended Mitigation Steps

When publishing a project, if the `lentAmount` for the community is non-zero, calculate the interest before updating the APR.

parv3213 (Rigor) confirmed

Medium Risk Findings (20)

[M-01] `Project.changeOrder()` would work unexpectedly for non SCConfirmed tasks.

Submitted by hansfriese

Project.sol#L402

The `changeOrder()` function will revert when it's called for the tasks that don't have subcontractors.

As a result, the project builder and contractor can't change the cost of a task until the subcontractor is confirmed.

Proof of Concept

The `changeOrder()` is used to change the cost or subcontractor of a task and there is no documentation that this function must be called only after a subcontractor is confirmed.

Also, it's reasonable to be able to change the cost when a subcontractor isn't confirmed yet.

But when it checks signature here, it assumes the task has a confirmed subcontractor already and `checkSignatureTask()` will revert in other cases.

So this function will be useless for non SCConfirmed tasks.

Tools Used

Solidity Visual Developer of VSCode

Recommended Mitigation Steps

We should check separately in case the subcontractor is confirmed or not here.

```
    if (getAlerts(_taskID)[2]) {
        // If subcontractor has confirmed.
        checkSignatureTask(_data, _signature, _taskID);
    } else {
        // If subcontractor not has confirmed.
        checkSignature(_data, _signature);
    }
```

parv3213 (Rigor) acknowledged

0xA5DF (warden) commented:

> I think this is invalid, since *currently* `checkSignatureTask` will pass if SC is the zero address and the signature isn't a valid signature (i.e. builder and GC can just pass zero as the signature).

> This will only be valid if the sponsor fixes other bugs by making `checkSignatureValidity()` revert on invalid signature.

Jack the Pug (judge) decreased severity to Medium and commented:

> Will downgrade to Medium given this is a feature being malfunctioning.

[M-02] Missing upper limit definition in `replaceLenderFee()` of `HomeFi.sol`

Submitted by MiloTruck, also found by __141345__, 0x52, 8olidity, cccz, Ch_301, codexploder, cryptonue, hansfriese, Ruhum, and sseefried

Community.sol#L392-L394
HomeFi.sol#L184-L197

The admin of the `HomeFi` contract can set `lenderFee` to greater than 100%, forcing calls to `lendToProject()` to all projects created in the future to revert.

Proof of Concept

Using the function `replaceLenderFee()`, admins of the `HomeFi` contract can set `lenderFee` to any arbitrary `uint256` value:

```
185:         function replaceLenderFee(uint256 _newLenderFee)
186:             external
187:             override
188:             onlyAdmin
189:         {
190:             // Revert if no change in lender fee
191:             require(lenderFee != _newLenderFee, "HomeFi::!C
192:
        // Reset variables
        lenderFee = _newLenderFee;

        emit LenderFeeReplaced(_newLenderFee);
    }
```

New projects that are created will then get its `lenderFee` from the `HomeFi` contract. When communities wish to lend to these projects, it calls `lendToProject()`, which has the following calculation:

```
    // Calculate lenderFee
    uint256 _lenderFee = (_lendingAmount * _projectInst
        (_projectInstance.lenderFee() + 1000);
```

If `lenderFee` a large value, such as `type(uint256).max`, the calculation shown above to overflow. This prevents any community from lending to any new projects.

Recommended Mitigation Steps

Consider adding a reasonable fee rate bounds checks in the `replaceLenderFee()` function. This would prevent potential griefing and increase the trust of users in the contract.

[zgorizzo69 (Rigor) confirmed]()

[M-03] Signature Checks could be passed when `SignatureDecoder.recoverKey()` returns 0

Submitted by cryptphi, also found by 0x1f8b and defsec

[Project.sol#L887]()
[Project.sol#L108-L115]()

It is possible to pass Signature Validity check with an SignatureDecoder.recoverKey() returns 0 whenever the builder and /or contractor have an existing approved hash for a data.

With occurrence of above, any user can call changeOrder or setComplete functions successfully after user approves data hashes.

Recommended Mitigation Steps

There should be a require check for `_recoveredSignature != 0` in `checkSignatureValidity()`.

[parv3213 (Rigor) confirmed]

[M-04] Hash approval not possible when contractor == subcontractor

Submitted by Lambda, also found by hansfriese

[Project.sol#L859]

When a contractor (let's say Bob) is also a subcontractor (which can be a valid scenario), it is not possible to use the hash approval feature for `checkSignatureTask`. The first call to `checkSignatureValidity` will already delete `approvedHashes[address(Bob)][_hash]`, the second call therefore fails.

Note that the same situation would also be possible for builder == contractor, or builder == subcontractor, although those situations are probably less likely to occur.

Recommended Mitigation Steps

Delete the approval only when all checks are done.

[parv3213 (Rigor) confirmed]

[M-05] Anyone can create disputes if `contractor` is not set

Submitted by berndartmueller, also found by 0xA5DF, arcoun, rotcivegaf, and wastewa

Project.sol#L498-L502
SignatureDecoder.sol#L25

Disputes enable an actor to arbitrate & potentially enforce requested state changes. However, the current implementation does not properly implement authorization, thus anyone is able to create disputes and spam the system with invalid disputes.

Proof of Concept

Calling the `Project.raiseDispute` function with an invalid `_signature`, for instance providing a `_signature` with a length of 66 will return `address(0)` as the recovered signer address.

Project.raiseDispute

```
    function raiseDispute(bytes calldata _data, bytes calldata _sigr
        external
        override
    {
        // Recover the signer from the signature
        address signer = SignatureDecoder.recoverKey(
            keccak256(_data),
            _signature,
            0
        );

        ...
      }
```

SignatureDecoder.sol#L25

```
    function recoverKey(
      bytes32 messageHash,
      bytes memory messageSignatures,
      uint256 pos
    ) internal pure returns (address) {
      if (messageSignatures.length % 65 != 0) {
          return (address(0));
      }
```

```
        ...
    }
```

If `_task` is set to `0` and the project does not have a `contractor`, the `require` checks will pass and `IDisputes(disputes).raiseDispute(_data, _signature);` is called. The same applies if a specific `_task` is given and if the task has a `subcontractor`. Then the check will also pass.

[Project.raiseDispute]

```
    function raiseDispute(bytes calldata _data, bytes calldata _sign
        external
        override
    {
        // Recover the signer from the signature
        address signer = SignatureDecoder.recoverKey(
            keccak256(_data),
            _signature,
            0
        );

        // Decode params from _data
        (address _project, uint256 _task, , , ) = abi.decode(
            _data,
            (address, uint256, uint8, bytes, bytes)
        );

        // Revert if decoded project address does not match this con
        require(_project == address(this), "Project::!projectAddress

        if (_task == 0) {
            // Revet if sender is not builder or contractor
            require(
                signer == builder || signer == contractor, // @audit
                "Project::!(GC||Builder)"
            );
        } else {
            // Revet if sender is not builder, contractor or task's
            require(
                signer == builder ||
                    signer == contractor || // @audit-info if `contr
                    signer == tasks[_task].subcontractor,
                "Project::!(GC||Builder||SC)"
            );

            if (signer == tasks[_task].subcontractor) {
                // If sender is task's subcontractor, revert if invi
                require(getAlerts(_task)[2], "Project::!SCConfirmed'
            }
        }

        // Make a call to Disputes contract raiseDisputes.
        IDisputes(disputes).raiseDispute(_data, _signature); // @auc
    }
```

Recommended Mitigation Steps

Consider checking the recovered `signer` address in `Project.raiseDispute` to not equal the zero-address:

```
    function raiseDispute(bytes calldata _data, bytes calldata _sigr
        external
        override
    {
        // Recover the signer from the signature
        address signer = SignatureDecoder.recoverKey(
            keccak256(_data),
            _signature,
            0
        );

        require(signer != address(0), "Zero-address"); // @audit-inf

        ...
      }
```

parv3213 (Rigor) confirmed

[M-06] Attacker can drain all the projects within minutes, if admin account has been exposed

Submitted by 0xA5DF, also found by Lambda and sseefried

HomeFi.sol#L156-L169
HomeFi.sol#L199-L208

In case where the admin wallet has been hacked, the attacker can drain all funds out of the project within minutes. All the attacker needs is the admin to sign a single meta/normal tx.
Even though the likelihood of the admin wallet being hacked might be low, given that the impact is critical - I think this makes it at least a medium bug.

Examples of cases where the attacker can gain access to admin wallet:

- The computer which the admins are using has been hacked
    - Even if a hardware wallet is used, the attacker can still replace the data sent to the wallet the next time the admin has to sign a tx (whether it's a meta or normal tx)
- The website/software where the meta tx data is generated has been hacked and attacker modifies the data for tx
- A malicious website tricks the admin into signing a meta tx to replace the admin or forwarder

Since the forwarder has the power to do everything in the system , once an attacker manages to replace it with a malicious forwarder, he can do whatever he wants withing minutes:

- The forwarder can replace the admin
- The forwarder can drain all funds from all projects by changing the subcontractor and marking tasks as complete, or adding new tasks / changing task cost as needed.

Even when signatures are required, you can bypass it by using the `approveHash` function.

Proof of Concept

Here's a PoC for taking over and running the `Project.setComplete()` function (I haven't included a whole process of changing SC etc. since that would be too time consuming, but there shouldn't be a difference between functions, all can be impersonated once you control the forwarder).

The PoC was added to projectTests.ts#L1109, and is based on the 'should be able to complete a task' test.

```
it('PoC forwarder overtake', async () => {
  const attacker = signers[10];


  // deploy the malicious forwarder
  const maliciousForwarder = await deploy<MaliciousForwarder>(
  const adminAddress = await homeFiContract.admin();
  const adminSigner = getSignerByAddress(signers, adminAddress
  // attacker takes over
  await homeFiContract.connect(adminSigner).setTrustedForwarde

  // attacker can now replace the admin, so that admin can't s
  let { data } = await homeFiContract.populateTransaction.repl
    attacker.address
  );
  let from = adminAddress;
  let to = homeFiContract.address;
  if (!data) {
    throw Error('No data');
  }
  let tx = await executeMetaTX(from, to, data);

  // assert that admin has been replaced by attacker
  expect(await homeFiContract.admin()).to.be.eq(attacker.addre

  // attacker can now execute setComplete() using the approveH

  const taskID = 1;
  const _taskCost = 2 * taskCost;
  const taskSC = signers[3];
  let completeData = {
    types: ['uint256', 'address'],
    values: [taskID, project.address],
  };
  const [encodedData, hash] = await encodeDataAndHash(complete
  await mockDAIContract.mock.transfer
    .withArgs(taskSC.address, _taskCost)
    .returns(true);
```

```
await mockDAIContract.mock.transfer
  .withArgs(await homeFiContract.treasury(), _taskCost / 1e3
  .returns(true);

({data} = await project.populateTransaction.approveHash(hash
let contractor = await project.contractor();
let {subcontractor} = await project.getTask(taskID);
let builder = await project.builder();

await executeMetaTX(contractor, project.address, data as str
await executeMetaTX(subcontractor, project.address, data as
await executeMetaTX(builder, project.address, data as string


tx = await project.setComplete(encodedData, "0x");
await tx.wait();

await expect(tx).to.emit(project, 'TaskComplete').withArgs(t

const { state } = await project.getTask(taskID);
expect(state).to.equal(3);
const getAlerts = await project.getAlerts(taskID);
expect(getAlerts[0]).to.equal(true);
expect(getAlerts[1]).to.equal(true);
expect(getAlerts[2]).to.equal(true);
expect(await project.lastAllocatedChangeOrderTask()).to.equa
expect(await project.changeOrderedTask()).to.deep.equal([]);

async function executeMetaTX(from: string, to: string, data:
  const gasLimit = await ethers.provider.estimateGas({
    to,
    from,
    data,
  });
  const message = {
    from,
    to,
    value: 0,
    gas: gasLimit.toNumber(),
    nonce: 0,
    data,
  };

  // @ts-ignore
  let tx = await maliciousForwarder.execute(message, "0x");
  return tx;
```

```
      }
    });


  // ----------------------------------------------------------- //
  // Added to ethersHelpers.ts file:
  export function encodeDataAndHash(
    data: any): string[] {
    const encodedData = encodeData(data);
    const encodedMsgHash = ethers.utils.keccak256(encodedData);
    return [encodedData, encodedMsgHash];
  }
```

Recommended Mitigation Steps

- Limit `approveHash` to contracts only - I understood from the sponsor that it is used for contracts to sign hashes. So limiting it to contracts only can help prevent stealing funds (from projects that are held by EOA) in case that the forwarder has been compromised (this is effective also in case there's some bug in the forwarder contract).

  - Alternately, you can also make it use `msg.sender` instead of `_msgSender()` , this will also have a similar effect (it will allow also EOA to use the function, but not via forwarder).

    - The advantage is that not only it wouldn't cost more than now, it'll even save gas.

    - Another advantage is that it will also protect projects held by contracts from being impersonated by a malicious forwarder

- Make the process of replacing the forwarder or the admin a 2 step process with a delay between the steps (except for disabling the forwarder, in case the forwarder was hacked). This will give the admin the option to take steps to stop the attack, or at least give the users time to withdraw their money.

```
    /// @inheritdoc IHomeFi
    function replaceAdmin(address _newAdmin)
        external
        override
        onlyAdmin
        nonZero(_newAdmin)
        noChange(admin, _newAdmin)
    {
        // Replace admin
        pendingAdmin = _newAdmin;

        adminReplacementTime = block.timestamp + 1 days;
        emit AdminReplaceProposed(_newAdmin);
    }

        /// @inheritdoc IHomeFi
    function executeReplaceAdmin()
        external
        override
        onlyAdmin

    {
        require(adminReplacementTime > 0 && block.timestamp > ac
        // Replace admin
        admin = pendingAdmin;

        emit AdminReplaced(_newAdmin);
    }
    /// @inheritdoc IHomeFi
    function setTrustedForwarder(address _newForwarder)
        external
        override
        onlyAdmin
        noChange(trustedForwarder, _newForwarder)
    {
        // allow disabling the forwarder immediately in case it
        if(_newForwarder == address(0)){
            trustedForwarder = _newForwarder;
        }
        forwarderSetTime = block.timestamp + 3 days;
        pendingTrustedForwarder = _newForwarder;
    }

    function executeSetTrustedForwarder(address _newForwarder)
        external
        override
        onlyAdmin
    {
        require(forwarderSetTime > 0 &&  block.timestamp > forwa
        trustedForwarder = pendingTrustedForwarder;
    }
```

- Consider removing the meta tx for `HomeFi` `onlyAdmin` modifier (i.e. usg `msg.sender` instead of `_msgSender()`), given that it's not going to be used that often it may be worth giving up the comfort for hardening security

parv3213 (Rigor) confirmed

0xA5DF (warden) commented:

> Dupe of @sseefried's **#165**

> Edit: On a second look issue 165 focuses more on not giving the forwarder the ability to impersonate the admin, and less on the damage that can be done with the forwarder using normal functionality (i.e. impersonating regular users, being able to drain all funds from projects).
> Also the suggested mitigation is very different.
> I think this makes this a different issue, but leaving this to the judge to decide.

Jack the Pug (judge) commented:

> Both this and 165 are good findings, I tend to merge 165 into this. The usage of EIP2771 is not very common, and I think you raised a noteworthy point that: a relayer's `_msgSender` is less trustworthy than the real `msg.sender`, the admin themself should not be trusted too much either.

> I also like your writing, short but comprehensive. Thanks for being part of the C4 community, @0xA5DF!

[M-07] `Project.raiseDispute()` doesn't use approvedHashes - meaning users who use contracts can't raise disputes

Submitted by 0xA5DF

Project.sol#L493-L536

In case users are using a contract (like a multisig wallet) to interact with a project, they can't raise a dispute.

The sponsors have added the `approveHash()` function to support users who wish to use contracts as builder/GC/SC. However, the `Project.raiseDispute()` function doesn't check them, meaning if any of those users wish to raise a dispute they can't do it.

Proof of Concept

I've modified the following test, trying to use an approved hash. The test failed.

```
    it('Builder can raise addTasks() dispute', async () => {
        let expected = 2;
        const actionValues = [
          [exampleHash],
          [100000000000],
          expected,
          projectAddress,
        ];
        // build and raise dispute transaction
        const [encodedData, signature] = await makeDispute(
          projectAddress,
          0,
          1,
          actionValues,
          signers[0],
          '0x4222',
        );
        const encodedMsgHash = ethers.utils.keccak256(encodedData)
        await project.connect(signers[0]).approveHash(encodedMsgHa
        let tx = await project
          .connect(signers[1])
          .raiseDispute(encodedData, "0x");
        // expect event
        await expect(tx)
          .to.emit(disputesContract, 'DisputeRaised')
          .withArgs(1, '0x4222');
        // expect dispute raise to store info
        const _dispute = await disputesContract.disputes(1);
        const decodedAction = abiCoder.decode(types.taskAdd, _disp
        expect(_dispute.status).to.be.equal(1);
        expect(_dispute.taskID).to.be.equal(0);
```

```
      expect(decodedAction[0][0]).to.be.equal(exampleHash);
      expect(decodedAction[1][0]).to.be.equal(100000000000);
      expect(decodedAction[2]).to.be.equal(expected);
      expect(decodedAction[3]).to.be.equal(projectAddress);
      // expect unchanged number of tasks
      let taskCount = await project.taskCount();
      expect(taskCount).to.be.equal(expected);
    });
```

Recommended Mitigation Steps

Make `raiseDispute()` to check for approvedHashes too.

parv3213 (Rigor) confirmed

Jack the Pug (judge) commented:

> Very nice!

[M-08] Builders must pay more interest when the system is paused.

Submitted by hansfriese, also found by 0x52, 0xNazgul, and rbserver

Community.sol#L455
Community.sol#L484
Community.sol#L509

Builders can't repay when the system is paused so they must pay more interest for the paused period.

Proof of Concept

Builders can repay to lenders using 3 functions, repayLender(), reduceDebt(), and escrow().

But they all don't work when the system is paused and builders have no way to avoid it.

Furthermore, the HomeFi admin is the main lender of builders and there is no assurance that the admin would pause the community for a while to get more interest.

Tools Used

Solidity Visual Developer of VSCode

Recommended Mitigation Steps

Recommend thinking of an approach to make 3 repay functions work for paused or modify the interest calculation formula not to add interest for the paused period.

zgorizzo69 (Rigor) acknowledged

parv3213 (Rigor) commented:

> The pause period is to fix severe bugs, and we don't want extra logic to handle extra interest. Hopefully, during that downtime, no builders will need to make repayment right away.
> Also, moving forward, HomeFi admin will be a decentralized DAO.

[M-09] It should not submit a project with no total budget. Requires at least one task with cost > 0

Submitted by cryptonue, also found by aez121, hansfriese, obront, rbserver, and saneryee

Community.sol#L206-L282

When publishing a project, there is still possibility the project doesn't have any task or 0 budget.

Proof of Concept

According to contest guideline, there is information that says:

> *"Note that you cannot submit a project with no total budget. Therefore it requires at least one task with a budget > 0."*

Meanwhile, on `publishProject()` in Community.sol, there is no check of this condition.

Recommended Mitigation Steps

Add a new `require` which will check if the first task (which is at index 1), its cost is > 0.

```
    // Local instance of variables. For saving gas.
    IProject _projectInstance = IProject(_project);
    ...

    // Revert if project doesn't have one task with budget > 0
    require(_projectInstance.tasks[1].cost > 0, "First task > 0");
```

parv3213 (Rigor) acknowledged and commented:

> The docs here were deprecated. A project doesn't have to have any task published in a community.

Jack the Pug (judge) commented:

> This is a valid Medium based on the docs (even though it's deprecated now).

[M-10] Possible DOS in `lendToProject()` and `toggleLendingNeeded()` function because unbounded loop can run out of gas

Submitted by minhquanym, also found by berndartmueller, Chom, and scaraven

Project.sol#L710

In `Project` contract, the `lendToProject()` function might not be available to be called if there are a lot of Task in `tasks[]` list of project. It means that the project cannot be funded by either builder or community owner.

This can happen because `lendToProject()` used `projectCost()` function. And the loop in `projectCost()` did not have a mechanism to stop, it's only based on the length `taskCount`, and may take all the gas limit. If the gas limit is reached, this transaction will fail or revert.

Same issue with `toggleLendingNeeded()` function which also call `projectCost()` function.

Proof of Concept

Function `projectCost()` did not have a mechanism to stop, only based on the `taskCount`.

```
    function projectCost() public view override returns (uint256 _co
        // Local instance of taskCount. To save gas.
        uint256 _length = taskCount;

        // Iterate over all tasks to sum their cost
        for (uint256 _taskID = 1; _taskID <= _length; _taskID++) {
            _cost += tasks[_taskID].cost;
        }
    }
```

There is no limit for builder when add task.

And function `lendToProject()` used `projectCost()` to check the new total lent value

```
    require(
        projectCost() >= uint256(_newTotalLent),
        "Project::value>required"
    );
```

Recommended Mitigation Steps

Consider keeping value of `projectCost()` in a storage variable and update it when a task is added or updated accordingly.

parv3213 (Rigor) confirmed

Jack the Pug (judge) decreased severity to Medium

[M-11] Owner of project NFT has no purpose

Submitted by berndartmueller, also found by byndooa and rbserver

Creating a new project mints a NFT to the `_sender` (builder). The `builder` of a project has special permissions and is required to perform various tasks.

However, if the minted NFT is transferred to a different address, the `builder` of a project stays the same and the new owner of the transferred NFT has no purpose and no permissions to access authorized functions in Rigor.

If real-world use-cases require a change of the `builder` address, there is currently no way to do so. Funds could be locked in the project contract if the current `builder` address is unable to perform any more actions.

Proof of Concept

HomeFi.sol#L225

```
function createProject(bytes memory _hash, address _currency)
    external
    override
    nonReentrant
{
    // Revert if currency not supported by HomeFi
    validCurrency(_currency);

    address _sender = _msgSender();

    // Create a new project Clone and mint a new NFT for it
    address _project = projectFactoryInstance.createProject(
        _currency,
        _sender
    );
    mintNFT(_sender, string(_hash));
```

```
        // Update project related mappings
        projects[projectCount] = _project;
        projectTokenId[_project] = projectCount;

        emit ProjectAdded(projectCount, _project, _sender, _currency
    }
```

Recommended Mitigation Steps

Consider preventing transferring the project NFT to a different address for now as long as there is no use-case for the NFT owner/holder or use the actual NFT owner as the `builder` of a project.

zgorizzo69 (Rigor) disputed and commented:

> Builders are kyc'ed that's why just by transferring the NFT you don't get any of the builder privileges.

parv3213 (Rigor) commented:

> As the warden said, `Owner of project NFT has no purpose` is true and is the intended behavior. Owning this NFT does not change anything.

Jack the Pug (judge) confirmed as valid

[M-12] `updateProjectHash` does not check project address

Submitted by MEP, also found by byndooa, Haipls, and minhquanym

Project.sol#L162

In Project.sol, function `updateProjectHash` L162, `_data` (which is signed by builder and/or contractor) does not contain a reference to the project address. In all other external functions of Project.sol, `_data` contains the address of the project, used in this check:

`require(_projectAddress == address(this), "Project::!projectAddress");`.

The lack of this verification makes it possible to reuse the same `_data`, and the same `_signature` on another project, in the case the latter has the same builder and/or contractor, and the same `_nonce`. In pratice, if the same group of people starts a new project, when `_nonce` reaches the correct value, anyone can change the hash of a task (if we suppose that that `updateTaskHash()` was used in the previous project).

parv3213 (Rigor) confirmed

[M-13] In `Project.setComplete()`, the signature can be reused when the first call is reverted for some reason

Submitted by hansfriese, also found by cccz

Project.sol#L330

`setComplete()` function might be called successfully using the past signature when it shouldn't work.

As a result, a task might be completed when a builder doesn't want it.

Proof of Concept

approveHash() function can set only true so there is no method to cancel already approved hash without passing validation here.

So the below scenario would be possible.

- A builder, GC, and SC started a task and SC finished the task.
- They are approved to complete the task and signed the signature.
- But right before to call setComplete() using the signature, the SC felt the cost is too low and raised a dispute to change the order using raiseDispute().
- As I suggested with another medium issue, the task can't be completed when there is an ongoing dispute from this document - "If there is no ongoing dispute about that project, task status is updated and payment is made.". So `setComplete()` might revert.

- Even if it doesn't check active disputes as now, `setComplete()` might revert when the funds haven't been allocated and a builder signed by fault.
- After that, the HomeFi admin accepted the dispute, and the cost of the task was increased as SC wanted.
- Then the builder would hope to get more results (or scores) from this task as the cost is increased rather than completed right away.
- But SC can call `setComplete()` using the previous signature and complete the task without additional work.
- A builder might know about that before and try to update task hash but it will revert because SC doesn't agree to updateTaskHash().
- In this case, it's logical to cancel the approved hash here but there is no such option.

I don't know if there would be similar problems with other functions that use signature and I think it would reduce the risk a little if we add an option to cancel the approved hash.

Tools Used

Solidity Visual Developer of VSCode

Recommended Mitigation Steps

Recommend modifying approveHash() like below.

```
function approveHash(bytes32 _hash, bool _bool) external overric
    address _sender = _msgSender();
    // Allowing anyone to sign, as its hard to add restrictions
    // Store _hash as signed for sender.
    approvedHashes[_sender][_hash] = _bool; //++++++++++++++++++

    emit ApproveHash(_hash, _sender, _bool); //+++++++++++++++++
}
```

I am not so sure that a similar scenario would be possible in the Community contract also and recommend to change both functions together.

parv3213 (Rigor) confirmed

[M-14] Incorrect initialization of smart contracts with Access Control issue

Submitted by Haipls, also found by byndooa, cryptphi, and TrungOre

HomeFiProxy.sol#L216-L230
Community.sol#L102-L119
DebtToken.sol#L43-L58
Disputes.sol#L74-L81
HomeFi.sol#L92-L120
Project.sol#L94-L105
ProjectFactory.sol#L45-L55

All next Impact depends on actions and attention from developers when deployed

- Loss of funds
- Failure of the protocol, with the need for redeploy
- Loss of control over protocol elements (some smart contracts)
- The possibility of replacing contracts and settings with harmful ones

And other things that come out of it...

Proof of Concept

For a proper understanding of Proof of Concept, you need to understand the following things:

1. Hardhat does not stop the process with a deploy and does not show failed transactions if they have occurred in some cases
2. Malicious agents can trace the protocol deployment transactions and insert their own transaction between them

Reason:

- During deploy TransparentUpgradeableProxy's initialize method for initializing contracts not called. The third parameter responsible for this is an empty string. This causes the initialization process itself to be **delayed**

- Contract initialization methods have no check over who calls them

Example ProjectFactory.sol#L45-L55

Also suitable for other contracts, strings are attached in Links to affected code

Examples of exploiting the vulnerability

Failure of the protocol, with the need for redeploy && Loss of control over protocol elements (some smart contracts):

1. User listen transaction in mempool, etherscan, transaction in block etc
2. Finds the moment of deployment and sends the transaction for setup his HomeFi address in Disputes contract: Just he call initialize method and put his _homeFi parameter
3. In the event that hardhat tracked a failed transaction, the deployment will stop and you will need to start over. If the hardhead misses it and the developers do not check the result and the setting, access to this part will be lost and fix is needed

Loss of funds:

1. User listen transaction in mempool, etherscan, transaction in block for listne when HomeFi will deployed
2. Send transaction for initialize HomeFi with his _treasury address
3. Transfer the admin ownership the right to the real address to divert the eyes
4. The address of the treasury remains with the attacker
5. The protocol fees (fee) will be transfered to the attacker's address until it is detected

Recommended Mitigation Steps

Carry out checks at the initialization stage or redesign the deployment process with the initialization of contracts during deployment

zgorizzo69 (Rigor) confirmed, but disagreed with severity and commented:

> About the reasons

- TransparentUpgradeableProxy third parameter is optionally initialized with `_data` as explained in {ERC1967Proxy-constructor}
- incorrect modifiers check that addresses are not address(0)
  about the possible exploit
  Interesting take on how the dark forest's creatures can harm the deployment process 👍
  however if a tx fails the whole deployment script stops but I think it is a good practice to indeed verify after each initialization

Jack the Pug (judge) decreased severity to Medium and commented:

> Valid, but gonna downgrade it to Medium as the impact is not that severe in practice.

> Btw, in response to the response about the 2nd reason:

> Contract initialization methods have no check over who calls them.
> incorrect modifiers check that addresses are not address(0)

> "no check over who calls them" means no access control. It can be called by anyone. It's not about the input validation.

[M-15] `Project.addTasks()` wouldn't work properly when it's called from disputes contract.

Submitted by hansfriese, also found by Lambda

Project.sol#L238

`addTasks()` function checks this require() to make sure `_taskCount` is correct.

But it might revert when this function is called after a dispute because it takes a certain time to resolve disputes and other tasks might be added meanwhile.

Proof of Concept

The below scenario would be possible.

- A project contains 10 active tasks(taskCount = 10) and a builder and contractor are going to add one more task.
- There were some disagreements between a builder and contractor so they raised a dispute with _taskCount = 10 using raiseDispute().
- Normally it would take a certain time(like 1 day or more) to resolve the dispute as it must be done by HomeFi owner.
- Meanwhile, if the builder and contractor need to add another task, they should set `_taskCount = 10` and `taskCount` will be 11 after addition here.
- After that, the HomeFi admin agreed to add a task with `_taskCount = 10`, but it will revert here.

So currently, the project builder and contractor shouldn't add new tasks to make their previous dispute valid.

I think it's reasonable to modify that they can add other tasks even though there is an active dispute.

Tools Used

Solidity Visual Developer of VSCode

Recommended Mitigation Steps

I think we can modify not to compare taskCount when it's called from disputes contract.

So we can modify this part like below.

```
    if (_msgSender() != disputes) {
        require(_taskCount == taskCount, "Project::!taskCount");
    }
    else {
        _taskCount = taskCount;
    }
```

parv3213 (Rigor) confirmed

[M-16] New subcontractor can be set for a SCConfirmed task without current subcontractor consent

Submitted by hyh, also found by hansfriese

Malicious builder/contractor can change the subcontractor for any task even if all the terms was agreed upon and work was started/finished, but the task wasn't set to completed yet, i.e. it's `SCConfirmed`, `getAlerts(_taskID)[2] == true`. This condition is not checked by inviteSC().

For example, a contractor can create a subcontractor of her own and front run valid setComplete() call with a sequence of `inviteSC(task, own_subcontractor) -> setComplete()` with a signatory from the `own_subcontractor`, stealing the task budget from the subcontractor who did the job. Contractor will not breach any duties with the community as the task will be done, while raiseDispute() will not work for a real subcontractor as the task record will be already changed.

Setting the severity to be high as this creates an attack vector to fully steal task budget from the subcontractor as at the moment of any valid setComplete() call the task budget belongs to subcontractor as the job completion is already verified by all the parties.

Proof of Concept

inviteSC() requires either builder or contractor to call for the change and verify nothing else:

Project.sol#L295-L316

```
    /// @inheritdoc IProject
    function inviteSC(uint256[] calldata _taskList, address[] ca
        external
        override
    {
        // Revert if sender is neither builder nor contractor.
        require(
            _msgSender() == builder || _msgSender() == contracto
            "Project::!Builder||!GC"
        );
        // Revert if taskList array length not equal to scList a
        uint256 _length = _taskList.length;
        require(_length == _scList.length, "Project::Lengths !ma

        // Invite subcontractor for each task.
        for (uint256 i = 0; i < _length; i++) {
            _inviteSC(_taskList[i], _scList[i], false);
        }

        emit MultipleSCInvited(_taskList, _scList);
    }
```

_inviteSC() only checks non-zero address and calls inviteSubcontractor():

Project.sol#L747-L762

```
    function _inviteSC(
        uint256 _taskID,
        address _sc,
        bool _emitEvent
    ) internal {
        // Revert if sc to invite is address 0
        require(_sc != address(0), "Project::0 address");

        // Internal call to tasks invite contractor
        tasks[_taskID].inviteSubcontractor(_sc);

        // If `_emitEvent` is true (called via changeOrder) ther
        if (_emitEvent) {
            emit SingleSCInvited(_taskID, _sc);
        }
    }
```

inviteSubcontractor() just sets the new value:

Tasks.sol#L106-L111

```
    function inviteSubcontractor(Task storage _self, address _sc
        internal
        onlyInactive(_self)
    {
        _self.subcontractor = _sc;
    }
```

Task is paid only on completion by setComplete():

Project.sol#L349-L356

```
            // Mark task as complete. Only works when task is active
            tasks[_taskID].setComplete();

            // Transfer funds to subcontractor.
            currency.safeTransfer(
                tasks[_taskID].subcontractor,
                tasks[_taskID].cost
            );
```

This way the absence of `getAlerts(_taskID)[2]` check and checkSignatureTask() call in inviteSC() provides a way for builder or contractor to steal task budget from a subcontractor.

Recommended Mitigation Steps

Consider calling checkSignatureTask() when `getAlerts(_taskID)[2]` is true, schematically:

Project.sol#L310-L313

```
            // Invite subcontractor for each task.
            for (uint256 i = 0; i < _length; i++) {
+               if (getAlerts(_taskList[i])[2])
+                   checkSignatureTask(_data_with_scList[i], _signat
                _inviteSC(_taskList[i], _scList[i], false);
            }
```

This approach is already implemented in changeOrder() where `_newSC` is a part of hash that has to be signed by all the parties:

Project.sol#L386-L403

```
    function changeOrder(bytes calldata _data, bytes calldata _s
        external
        override
        nonReentrant
    {
        // Decode params from _data
        (
            uint256 _taskID,
            address _newSC,
            uint256 _newCost,
            address _project
        ) = abi.decode(_data, (uint256, address, uint256, addres

        // If the sender is disputes contract, then do not check
        if (_msgSender() != disputes) {
            // Check for required signatures.
            checkSignatureTask(_data, _signature, _taskID);
        }
```

[Project.sol#L477-L481]()

```
            // If new subcontractor is not zero address.
            if (_newSC != address(0)) {
                // Invite the new subcontractor for the task.
                _inviteSC(_taskID, _newSC, true);
            }
```

checkSignatureTask() checks all the signatures:

[Project.sol#L855-L861]()

```
            // When builder has not delegated rights to contract
            else {
                // Check for B, SC and GC signatures
                checkSignatureValidity(builder, _hash, _signatur
                checkSignatureValidity(contractor, _hash, _signa
                checkSignatureValidity(_sc, _hash, _signature, 2
            }
```

zgorizzo69 (Rigor) disputed and commented:

> When a SC accepts an invitation the task is marked as active Tasks.sol#L128.
> So as you noted here above the inviteSubcontractor for the same task will fail because of the modifier.

```
    function inviteSubcontractor(Task storage _self, address _sc
        internal
        onlyInactive(_self)
    {
```

hyh (warden) commented:

> Yes, you are right, in general case `onlyInactive` modifier guards the reset.
> The issue appears to be more specific, in the case when task budget is increased, while there is no budget to cover it, i.e. `totalLent - _totalAllocated < _newCost - _taskCost`, the subcontractor signs only the budget increase itself, while subcontractor ends up being unassigned from it fully:

> Project.sol#L422-L461

```
            // If tasks are already allocated with old cost.
            if (tasks[_taskID].alerts[1]) {
                // If new task cost is less than old task cost.
                if (_newCost < _taskCost) {
                    // Find the difference between old - new.
                    uint256 _withdrawDifference = _taskCost - _r

                    // Reduce this difference from total cost al
                    // As the same task is now allocated with le
                    totalAllocated -= _withdrawDifference;

                    // Withdraw the difference back to builder's
                    // As this additional amount may not be requ
                    autoWithdraw(_withdrawDifference);
                }
                // If new cost is more than task cost but total
                else if (totalLent - _totalAllocated >= _newCost
                    // Increase the difference of new cost and o
                    totalAllocated += _newCost - _taskCost;
            }
            // If new cost is more than task cost and totalI
            else {
                // Un-confirm SC, mark task as inactive, mar

                // Mark task as inactive by unapproving subc
                // As subcontractor can only be approved if
                _unapproved = true;
                tasks[_taskID].unApprove();

                // Mark task as not allocated.
                tasks[_taskID].unAllocateFunds();

                // Reduce total allocation by old task cost.
                // As as needs to go though funding process
                totalAllocated -= _taskCost;

                // Add this task to _changeOrderedTask array
                _changeOrderedTask.push(_taskID);
            }
        }
```

> Suppose task is 95% complete, its budget is fully spent, so changeOrder() is called per mutual agreement to add extra `0.05 * old_cost / 0.95` funds, which aren't lent yet. Dishonest contractor can call `inviteSC` with own subcontractor, who will receive full `old_cost / 0.95` on completion.

> I.e. fully removing subcontractor from already funded and started task provides a more specific similar attack surface.

> By definition `unApprove` deals with the case of new task that needs to be reviewed:

> [Tasks.sol#L153-L164]()

```
    /**
     * @dev Set a task as un accepted/approved for SC

     * @dev modifier onlyActive

     * @param _self Task the task being set as funded
     */
    function unApprove(Task storage _self) internal {
        // State/ lifecycle //
        _self.alerts[uint256(Lifecycle.SCConfirmed)] = false;
        _self.state = TaskStatus.Inactive;
    }
```

> But in changeOrder() all the parties already reviewed and accepted the terms:

> Project.sol#L391-L403

```
        // Decode params from _data
        (
            uint256 _taskID,
            address _newSC,
            uint256 _newCost,
            address _project
        ) = abi.decode(_data, (uint256, address, uint256, addres

        // If the sender is disputes contract, then do not check
        if (_msgSender() != disputes) {
            // Check for required signatures.
            checkSignatureTask(_data, _signature, _taskID);
        }
```

> Project.sol#L855-L861

```
            // When builder has not delegated rights to contract
            else {
                // Check for B, SC and GC signatures
                checkSignatureValidity(builder, _hash, _signatur
                checkSignatureValidity(contractor, _hash, _signa
                checkSignatureValidity(_sc, _hash, _signature, 2
            }
```

> So, marking the task as not active and not SCConfirmed doesn’t look correct in this case.

> Straightforward mitigation here is to keep it active, i.e. do partial flag removal, say do `unConfirm` instead of `unApprove`:

> Tasks.sol#L160-L164

```
    function unConfirm(Task storage _self) internal {
        // State/ lifecycle //
        _self.alerts[uint256(Lifecycle.SCConfirmed)] = false;
    }
```

hyh (warden) commented:

> A little bit more complex, but more correct (project logic aligned) mitigation is:

1. Introduce `deActivate` instead of `unConfirm`:

```
    function deActivate(Task storage _self) internal {
        // State/ lifecycle //
        _self.state = TaskStatus.Inactive;
    }
```

2. Introduce `onlyUnconfirmed` modifier and set it to the inviteSubcontractor() and acceptInvitation():

```
    /// @dev only allow unconfirmed tasks.
    modifier onlyUnconfirmed(Task storage _self) {
        require(
            !_self.alerts[uint256(Lifecycle.SCConfirmed)],
            "Task::SCConfirmed"
        );
        _;
    }
```

> Tasks.sol#L106-L111

```
    function inviteSubcontractor(Task storage _self, address _sc
        internal
-       onlyInactive(_self)
```

```
+         onlyUnconfirmed(_self)
      {
          _self.subcontractor = _sc;
      }
```

Tasks.sol#L119-L129

```
      function acceptInvitation(Task storage _self, address _sc)
          internal
-         onlyInactive(_self)
+         onlyUnconfirmed(_self)
      {
          // Prerequisites //
          require(_self.subcontractor == _sc, "Task::!SC");

          // State/ lifecycle //
          _self.alerts[uint256(Lifecycle.SCConfirmed)] = true;
          _self.state = TaskStatus.Active;
      }
```

`onlyInactive` can then be removed:

Tasks.sol#L42-L46

```
      /// @dev only allow inactive tasks. Task is inactive if SC i
      modifier onlyInactive(Task storage _self) {
          require(_self.state == TaskStatus.Inactive, "Task::activ
          _;
      }
```

3. Deactivate task only instead of fully resetting it in changeOrder():

Project.sol#L443-L460

```
                      else {
-                         // Un-confirm SC, mark task as inactive, mark

-                         // Mark task as inactive by unapproving subco
-                         // As subcontractor can only be approved if t
-                           _unapproved = true;
-                           tasks[_taskID].unApprove();

+                           // Mark task as inactive, mark allocated as f

+                           // Mark task as inactive
+                           tasks[_taskID].deActivate();

                            // Mark task as not allocated.
                            tasks[_taskID].unAllocateFunds();

                            // Reduce total allocation by old task cost.
                            // As as needs to go though funding process
                            totalAllocated -= _taskCost;

                            // Add this task to _changeOrderedTask array
                            _changeOrderedTask.push(_taskID);
                        }
```

> Notice that the mitigation here is to make Active and SCConfirmed states independent (as a general note, it doesn't make much sense to have some fully coinciding states). Active flags whether task is in progress right now, while SCConfirmed flags whether it ever was started, being either Active (work is being done right now) or Inactive (work had started, something was done, now it's paused).

> The issue basically means that the states are different and moving a task to another SC while it's SCConfirmed should be prohibited as some work was done and some payment to current SC is due

parv3213 (Rigor) confirmed, but disagreed with severity and commented:

> Agree to the risk, but the severity should be 2.

Jack the Pug (judge) decreased severity to Medium and commented:

> This is a good one with a very detailed explanation, but I'm afraid it fits a Medium severity better, as funds are not directly at risk but rather a malfunctioning feature that can indirectly cause damage to certain roles.

[M-17] Malicious delegated contractor can block funding tasks or mark tasks as complete

Submitted by indijanc

Project.sol#L219
Project.sol#L655
Project.sol#L807

A malicious delegated contractor can add a huge number of tasks (or one task with a huge cost). This would then pose problems in `allocateFunds()` as tasks could not be funded. Builder could remove delegation for the contractor but couldn't replace the contractor and so couldn't remove the malicious contractor. The contractor is required to sign various state changes in `Project.sol`. A delegated contractor can also for example complete tasks which results in transferring funds to subcontractors.

This sounds very problematic and would be critical, but reading through the documentation and the code, I'm assuming there is certain trust incorporated and required for the system to work. Hence I'm assuming the system considers a delegated contractor is trustworthy as is the builder. So while the impact may be big I consider the likelihood quite small.

Proof of Concept

When a contractor is delegated, various operations only need his signature.
Project.sol L807

Tools Used

Visual Studio Code

Recommended Mitigation Steps

There's a couple of improvements you could consider:

1. Create a function to update `lastAllocatedTask`. This could be restricted to `Disputes` contract or the builder. This could be used against maliciously inserted tasks.

2. Add functionality for `Disputes` contract to be able to remove or replace the contractor. This would be a guard against malicious contractors.

parv3213 (Rigor) acknowledged, but disagreed with severity

Jack the Pug (judge) commented:

> I like this finding, but this is probably a design choice. The suggestions make sense to me. I'll keep this as a Medium.

[M-18] Task Functionality completely sidestepped via `autoWithdraw`

Submitted by GalloDaSballo

Project.sol#L770

`autoWithdraw` will send funds to the `builder`, we can use this knowledge to drain all funds from `Project` to the builder contract. Completely sidestepping the whole Task based logic.

Impact

Through creation and deletion of tasks, leveraging `autoWithdraw` which will always send the funds to be `builder`, even when origin was the Community, a builder can cycle out all funds out of the Project Contract and transfer them to themselves.

Ultimately this breaks the trust assumptions and guarantees of the Task System, as the builder can now act as they please, the Project contract no longer holding any funds is limited.

Only aspect that diminishes impact is that the system is based on Credit (uncollateralized /undercollateralized lending), meaning the Builder wouldn't be "committing a crime" in taking ownership of all funds.

However the system invariants used to offer completely transparent accounting are now bypassed in favour of "trusting the builder".

Proof of Concept

We know we can trigger `autoWithdraw` it by creating and allocating a task, and then reducing it's cost

```
                // If tasks are already allocated with old cost.
                if (tasks[_taskID].alerts[1]) {
                    // If new task cost is less than old task cost.
                    if (_newCost < _taskCost) {
                        // Find the difference between old - new.
                        uint256 _withdrawDifference = _taskCost - _r

                        // Reduce this difference from total cost al
                        // As the same task is now allocated with le
                        totalAllocated -= _withdrawDifference;

                        // Withdraw the difference back to builder's
                        // As this additional amount may not be requ
                        autoWithdraw(_withdrawDifference);
                    } else if (totalLent - _totalAllocated >= _newCo
```

To funnel the funds we can:

- Create a new Task with Cost X (call `addTasks`)
- Allocate to it (call `allocateFunds`)
- `changeOrder` to trigger the condition `if (_newCost < _taskCost) {` and receive the delta of tokens

Repeat until all funds are funneled into the `builder` wallet.

The reason why the builder can do this is because in all functions involved:

- `addTasks`
- `changeOrder`

only the `builder` signature is necessary, meaning the contract is fully trusting the `builder`

Example Scenario

- Builder funnels the funds out
- Builder makes announcement: “Funds are safu, we’ll update once we know what to do next”
- Builder follows up: “We will use twitter to post updates on the project”
- Entire system is back to being opaque, making the system pointless

Recommended Mitigation Steps

Below are listed two options for mitigation

- A) Consider removing `autoWithdraw` (keep funds inside of project), create a separate multi-sig like way to withdraw
- B) Keep a split between funds sent by Builder and by Community, and make `autoWithdraw` send the funds back accordingly (may also need to re-compute total sent in Community)

parv3213 (Rigor) acknowledged and commented:

> Users in our system are KYC’ed, whitelisted, and trusted. We are certain that they won’t misuse this feature.

Jack the Pug (judge) decreased severity to Medium and commented:

> The issue makes a lot of sense to me, from the security perspective, the system should have as minimal trust as possible. The recommended remediation also makes sense.

> I’m not sure about the High severity though. It’s more like a Medium to me.

[M-19] `changeOrder` requires subcontractor signature when the subcontractor address is 0

Submitted by Lambda

Project.sol#L402
Project.sol#L485

Via `changeOrder`, it is possible to set the subcontractor address to 0 (and it is zero when no one is invited). However, when it is updated later again, a signature of the "current subcontractor" (in this case `address(0)`) is still required. This is in contrast to contractors, where the signature is only required when the contractor address is non-zero.

Proof Of Concept

1.) Task 1 is assigned to the subcontractor Bob.
2.) `changeOrder` with Bob's signature is used to assign task 1 temporarily to address 0 while a new subcontractor is searched.
3.) The price of the task should be changed, which requires the signature of the "current subcontractor" (i.e., `address(0)`)

To be fair, because `SignatureDecoder.recoverKey` returns `address(0)` for invalid signatures, an invalid signature could in theory be submitted in step 3. But I do not assume that this is really intended (for instance, there is also the check in `checkSignatureTask`, although one could simply use an invalid signature when it is `address(0)`) and a design that requires the user to submit invalid signatures in certain scenarios would also be very poor in my opinion.

Recommended Mitigation Steps

Check if the subcontractor address is zero, do not require a valid signature in such cases.

parv3213 (Rigor) acknowledged

[M-20] `Project.sol` and `Community.sol` have no way to revoke a hash in approvedHashes

Submitted by 0x52

Community.sol#L501-L506
Project.sol#L108-L115

User is unable to revoke previously approved hash.

Proof of Concept

If user reconsiders or notices something malicious about the hash after signing, they should be able to revoke the hash. For example the user approves a hash only to find out later that the hash has been spoofed and they weren't approving what they thought they were. To protect themselves the user should be able to revoke approval, otherwise it may lead to loss of funds or access.

Recommended Mitigation Steps

Add the following function:

```
    function revokeHash(bytes32 _hash) external virtual {
        approvedHashes[_msgSender()][_hash] = false;
    }
```

parv3213 (Rigor) disputed and commented:

> I do not find it essential to revoke a hash. As off-chain signatures can never be marked as invalid, adding this feature for on-chain signatures makes no sense.

Low Risk and Non-Critical Issues

For this contest, 99 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Lambda** received the top score from the judge.

The following wardens also submitted reports: GalloDaSballo, Deivitto, 0xA5DF, 0xNazgul, Guardian, IllIllI, joestakey, MEP, rokinot, saian, 0x1f8b, scaraven, dipp, Dravee, ElKu, horsefacts, hyh, neumo, oyc_109, Ruhum, simon135, sseefried, rbserver, JC, minhquanym, pfapostol, Rolezn, hansfriese, 0x52, benbaessler, brgltd, c3phas, Extropy, samruna, Bahurum, bobirichman, mics, sikorico, Chom, CodingNameKiki, ReyAdmirado, robee, SooYa, cryptonue, ak1, CertoraInc, defsec, hake, TrungOre, _Adam, 0xf15ers, Bnke0x0, bulej93, Funen, Soosh, byndooa, __141345__, 0xNineDec, exd0tpy, berndartmueller, ajtra, Aymen0909, codexploder, ignacio, indijanc, Rohan16, Sm4rty, supernova, TomJ, Waze, cryptphi, obront, arcoun, asutorufos, ayeslick, bin2chen, erictee, fatherOfBlocks, Noah3o6, rotcivegaf, 8olidity, 0xkatana, 0xSmartContract, 0xSolus, delfin454000, gogo, kaden, poirots, sach1r0, Yiko, 0xsolstars, a12jmx, djxploit, p_crypt0, saneryee, Tomio, Jujic, *and* Throne6g.

[01]

`isPublishedToCommunity` does not check that the `_communityID` exists. It can return true for non-published projects by passing in a `_communityID` of 0. This enables for instance to call `unpublishProject` on unpublished projects (or paying the publish fee for a non-existing project with `_communityID = 0`. While this is not a major issue, it can be confusing (because events are emitted) and building upon this modifier in the future can be dangerous. Consider validating the `_communityID`.

[02]

In `escrow`, it is possible that the `_agent` is the zero address, in which case signature validation succeeds with any invalid signature (i.e., no actual escrow, as there is no agent). Consider adding a check that the `_agent` is non-zero.

[03]

In `Community`, adding members and updating the community hash is possible when the system is changed. As these also change the system state, consider also requiring that the system is not paused.

[04]

There is no way to remove members of a community (e.g., misbehaving members), which might be desirable.

[05]

The comment "// Burn _interestEarned amount wrapped token to lender" is wrong Community.sol#L849, this should be mint instead of burn.

[06]

In general, it is considered good practice to provide a deadline for signatures and a way to revoke them (e.g., when a private key is compromised), which is both currently not implemented.

[07]

In `changeOrder`, it is not checked if the task actually exists. While changing the cost for a non-existing task is not possible (because of the `getAlerts` check), the owner can be set: First, the task will be unapproved, setting the status to inactive. Then, the subcontractor is invited, which succeeds, as the task is inactive. The subcontractor can even accept the invitation, which marks the task as active, although he was never created / initialized. Consider adding a check if the task already exists.

[08]

`raiseDispute` does not include any replay protection, meaning that anyone can raise the same dispute again after one was submitted.

[09]

It is possible to raise disputes for not (yet) existing tasks in `raiseDispute`, which should not be possible

[10]

It is mentioned that “This can be useful when trying to deploy new version of HomeFiProxy”. However, there is currently no clean way to do this. When a new HomeFi proxy is deployed and initialized, new proxies are deployed (i.e., the state is lost). Consider adding a way to initialize a new proxy with already existing proxy addresses.

[11]

`SignatureDecoder.recoverKey` does not support EIP-1271, meaning there is no support for smart contracts in all places that use signatures (which are many), which hinders different applications (e.g., building on top of the protocol).

[12]

The number of currencies (3) is hard-coded in different places, consider storing this information in arrays, which enables easy additions of new ones.

[13]

There is no upper limit for the lender fee HomeFi.sol#L194. Consider enforcing a limit of 1,000 (or even something like 200) to avoid errors and give users an upper limit for the fee.

[14]

`initiateHomeFi` is documented with "Can only be called by HomeFiProxy owner", but this is not true. The function is callable by anyone and sets the owner.

[15]

`recoverTokens` has a hardcoded 3 Project.sol#L369 instead of using the enum value, which can lead to problems when updating the possible enum values.

[16]

`checkPrecision` does not take the number of decimals into account. For USDC with 6 decimals means rounding to 0.1 pennies, whereas the precision is much higher (probably too high, which you want to avoid) for DAI with 18 decimals.

Jack the Pug (judge) commented:

> One of the best QA reports! Pure good findings found by keen human eyes. Good job!

parv3213 (Rigor) commented:

> [01] seems invalid as community id starts from 1. Community id 0 is always invalid.

Jack the Pug (judge) commented:

> Re: [01], I believe the issue is a valid low-severity issue, while it's true that REAL community id starts from 1, it still can not prevent the caller to use 0 as the community id and the unexpected behavior will happen if they do so, as described in the QA report. I don't think this requires a fix though, it's a minor issue indeed.

Gas Optimizations

For this contest, 91 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **c3phas** received the top score from the judge.

The following wardens also submitted reports: ElKu, Dravee, 0x1f8b, CertoraInc, fatherOfBlocks, GalloDaSballo, hyh, 0xkatana, 0xNazgul, benbaessler, defsec, gogo, hake, JC, MiloTruck, NoamYakov, IllIIII, __141345__, Chinmay, Deivitto, eierina, jag, oyc_109, pfapostol, ReyAdmirado, Ruhum, saian, samruna, TomJ, 0xSmartContract, Aymen0909, bharg4v, brgltd, delfin454000, Rolezn, _Adam, 0x040, ak1, ballx, durianSausage, erictee, Metatron, rbserver, ret2basic, 0xsam, ajtra, Bnke0x0, Fitraldys, gerdusx, kyteg, mics, simon135, Tomio, apostle0x01, lucacez, Rohan16, sach1r0, Sm4rty, Waze, 0xSolus, asutorufos, Chom, Funen, MEP, robee, SooYa, CodingNameKiki, Guardian, kaden, 8olidity, a12jmx, bobirichman, cryptonue, Extropy, ignacio, Noah3o6, dharma09, djxploit, PaludoX0, 0xA5DF, minhquanym, rokinot, scaraven, supernova, zeesaw, 0xc0ffEE, Lambda, sikorico, tofunmi, *and* teddav.

NB: *Some functions have been truncated where neccessary to just show affected parts of the code*

[G-01] Cache storage values in memory to minimize SLOADs

The code can be optimized by minimising the number of SLOADs. SLOADs are expensive 100 gas compared to MLOADs/MSTOREs(3gas)
Storage value should get cached in memory

File: HomeFi.sol Line 228

HomeFi.sol.createProject(): projectCount should be cached(Saves ~ 71 gas)

Average gas before caching = 339543
Average gas after caching = 339472

```
        function createProject(bytes memory _hash, address _currency
            external
            override
            nonReentrant
        {
            // Revert if currency not supported by HomeFi
            validCurrency(_currency);
       // Update project related mappings
            projects[projectCount] = _project;  @audit: SLOAD 1
            projectTokenId[_project] = projectCount; @audit: SLOAD 2


            emit ProjectAdded(projectCount, _project, _sender, _curr
        }
```

projectCount is being read 3 times in the following lines

SLOAD 1 Line 228
SLOAD 2 Line 229
SLOAD 3 Line 231

File: HomeFi.sol Line 284-297

HomeFi.sol.mintNFT(): projectCount should be cached

```
        function mintNFT(address _to, string memory _tokenURI)
            internal
            returns (uint256)
        {
            // Project count starts from 1
            projectCount += 1;


            // Mints NFT and set token URI
            _mint(_to, projectCount);
            _setTokenURI(projectCount, _tokenURI);


            emit NftCreated(projectCount, _to);
            return projectCount;
```

```
    }
```

File: Project.sol Line 176&179

Project.sol.updateProjectHash(): hashChangeNonce should be cached (saves ~ 101 gas)

Average gas before caching = 54538

Average gas after caching = 54437

```
    function updateProjectHash(bytes calldata _data, bytes callc
        external
        override
    {
        // Revert if decoded nonce is incorrect. This indicates
        require(_nonce == hashChangeNonce, "Project::!Nonce"); (
        // Increment to ensure a set of data and signature canno
        hashChangeNonce += 1;@audit - SLOAD 2 and
        emit HashUpdated(_hash);
    }
```

In the above function, there are two SLOADS that can be replaced with a cached variable.

SLOAD 1: Line 176

SLOAD 2: Line 179

File: Project.sol Line 277&290

Project.sol.updateTaskHash(): hashChangeNonce should be cached(saves ~ 98 gas)

Average gas before caching = 58185

Average gas after caching = 58087

```
    function updateTaskHash(bytes calldata _data, bytes calldata
        external
        override
    {
        // Decode params from _data
        (bytes memory _taskHash, uint256 _nonce, uint256 _taskII
            _data,
            (bytes, uint256, uint256)
        );
        // Revert if decoded nonce is incorrect. This indicates
        require(_nonce == hashChangeNonce, "Project::!Nonce");
       // Increment to ensure a set of data and signature cannot
        hashChangeNonce += 1;
        emit TaskHashUpdated(_taskID, _taskHash);
    }
```

In the above function, there are two SLOADS that can be replaced with a cached variable.
SLOAD 1: Line 277
SLOAD 2: Line 290

File: Project.sol Line 591-604

Project.sol.allocateFunds():_changeOrderedTask.length should be cached(Saves ~ 118 gas)

Average gas before caching = 63493
Average gas after caching = 63295

```
        uint256[] memory _tasksAllocated = new uint256[](
            taskCount - j + _changeOrderedTask.length - i @audit
        );
    // Number of times a loop has run.
        uint256 _loopCount;
        /// CHANGE ORDERED TASK FUNDING ///
        // Any tasks added to _changeOrderedTask will be allocat
        if (_changeOrderedTask.length > 0) { @audit - SLOAD 2
            // Loop from lastAllocatedChangeOrderTask to _change
            for (; i < _changeOrderedTask.length; i++) { @audit
                // Local instance of task cost. To save gas.

                //truncated a big chunk of code here
635:   if (i == _changeOrderedTask.length) { @audit - another SI
```

SLOAD 1: Line 592
SLOAD 2: Line 610
SLOAD 3: Read inside a for loop Line 603

SLOAD 4: Line 635
In the above function, the gas estimate might be higher than indicated due the SLOAD inside the for loop

File: Community.sol Line 143 & 150

Community.sol.createCommunity():communityCount should be cached(Saves ~186 gas)

Average gas before caching = 176852
Average gas after caching = 176666

```
140: communityCount++; //@audit - SLOAD 1 + SSTORE(communityC


     // Store community details
143: CommunityStruct storage _community = _communities[commun
     _community.owner = _sender;
     _community.currency = IDebtToken(_currency);
                    ...
      @ audit - Truncated some bit of code here
150: emit CommunityAdded(communityCount, _sender, _currency,
 }
```

SLOAD 1: Line 140
SLOAD 2: Line 143
SLOAD 3: Line 150
Note, after creating a **temp** variable in the above , for line 140, after incrementing the temp variable we need to assign the temp value to **communityCount**

[G-02] Cache the length of arrays in loops

The solidity compiler will always read the length of the array during each iteration. That is,

1.if it is a storage array, this is an extra sload operation (**100 additional extra gas (EIP-2929 2) for each iteration except for the first**),
2.if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first),

3.if it is a calldata array, this is an extra calldataload operation (3 additional gas for each iteration except for the first)

This extra costs can be avoided by caching the array length (in stack):

Here, I suggest storing the array's length in a variable before the for-loop, and use it instead:

File: Project.sol Line 603

Project.sol.allocateFunds(): _changeOrderedTask.length should be cached - _changeOrderedTask is a storage array

```
                for (; i < _changeOrderedTask.length; i++) {
```

This optimization is especially important if it is a storage array as it's our case here.

The above should be modified to

```
            uint256 length = _changeOrderedTask.length;
                for (; i < length; i++) {
```

[G-03] ++i costs less gas compared to i++ or i += 1 in for loops (~5 gas per iteration)

++i costs less gas compared to i++ or i += 1 for unsigned integer, as pre-increment is cheaper (about 5 gas per iteration). This statement is true even with the optimizer enabled.

i++ increments i and returns the initial value of i. Which means:

```
uint i = 1;
i++; // == 1 but i == 2
```

But ++i returns the actual incremented value:

```
    uint i = 1;
    ++i; // == 2 and i == 2 too, so no need for a temporary variable
```

In the first case, the compiler has to create a temporary variable (when used) for returning 1 instead of 2

Instances include:

File: HomeFiProxy.sol line 87

```
            for (uint256 i = 0; i < _length; i++) {
                _generateProxy(allContractNames[i], _implementations
            }
```

File: HomeFiProxy.sol line 136

```
            for (uint256 i = 0; i < _length; i++) {
                _replaceImplementation(_contractNames[i], _contractA
            }
```

File: Project.sol Line 248

```
            for (uint256 i = 0; i < _length; i++) {
```

File: Project.sol Line 311

```
            for (uint256 i = 0; i < _length; i++) {
                _inviteSC(_taskList[i], _scList[i], false);
            }
```

File: Project.sol Line 322

```
            for (uint256 i = 0; i < _length; i++) {
```

File: Tasks.sol Line 181

```
for (uint256 i = 0; i < _length; i++) _alerts[i] = _self
```

[G-04] ++x is more efficient than x++(Saves ~6 gas)

File: Community.sol Line 140
Average gas when using communityCount++ : 176852
Average gas when using ++communityCount : 176846

```
communityCount++;
```

Other Instances
File: Disputes.sol Line 121

```
emit DisputeRaised(disputeCount++, _reason);
```

[G-05] Splitting require() statements that use && saves gas - (saves 8 gas per &&)

Instead of using the && operator in a single require statement to check multiple conditions,using multiple require statements with 1 condition per require statement will save 8 GAS per &&
The gas difference would only be realized if the revert condition is realized(met).

File: Disputes.sol Line 61

```
require(
    _disputeID < disputeCount &&
        disputes[_disputeID].status == Status.Active,
    "Disputes::!Resolvable"
);
```

The above should be modified to

```
        require( _disputeID < disputeCount,  "Disputes::!Resolva
        require(disputes[_disputeID].status == Status.Active, "I
```

File: Disputes.sol Line 106

```
        require(
            _actionType > 0 && _actionType <= uint8(ActionType.T
            "Disputes::!ActionType"
        );
```

File: Community.sol Line 353

```
        require(
            _lendingNeeded >= _communityProject.totalLent &&
                _lendingNeeded <= IProject(_project).projectCost
            "Community::invalid lending"
        );
```

Proof
The following tests were carried out in remix with both optimization turned on and off

```
function multiple (uint a) public pure returns (uint){
        require ( a > 1 && a < 5, "Initialized");
        return  a + 2;
}
```

Execution cost
21617 with optimization and using &&
21976 without optimization and using &&

After splitting the require statement

```
function multiple(uint a) public pure returns (uint){
        require (a > 1 ,"Initialized");
        require (a < 5 , "Initialized");
```

```
            return a + 2;
    }
```

Execution cost
21609 with optimization and split require
21968 without optimization and using split require

[G-06] Comparisons: != is more efficient than > in require (6 gas less)

!= 0 costs less gas compared to > 0 for unsigned integers in require statements with the optimizer enabled (6 gas)

For uints the minimum value would be 0 and never a negative value. Since it cannot be a negative value, then the check > 0 is essentially checking that the value is not equal to 0 therefore >0 can be replaced with !=0 which saves gas.

Proof: While it may seem that > 0 is cheaper than !=, this is only true without the optimizer enabled and outside a require statement. If you enable the optimizer at 10k AND you're in a require statement, this will save gas. You can see this tweet for more proofs: https://twitter.com/gzeon/status/1485428085885640706

I suggest changing > 0 with != 0 here:

File: Project.sol Line 195

```
            require(_cost > 0, "Project::!value>0");
```

File: Community.sol Line 764

```
            require(_repayAmount > 0, "Community::!repay");
```

[G-07] Emitting storage values instead of the memory one(saves ~101 gas)

Here, the values emitted shouldn't be read from storage. The existing memory values should be used instead:

File: Project.sol Line 144 average gas while using the storage value - 69561 average gas while using the memory value - 69460

```
            // Store new contractor
            contractor = _contractor;
            contractorConfirmed = true;

            // Check signature for builder and contractor
            checkSignature(_data, _signature);

            emit ContractorInvited(contractor);@audit - should emit
        }
```

In the above we should emit **_contractor**

[G-08] Using unchecked blocks to save gas

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an unchecked block

File: Project.sol Line 427

```
            uint256 _withdrawDifference = _taskCost - _newCost;
```

The above operation cannot underflow due to the check on Line 425 which ensures that `_taskCost` is greater than `_newCost` before the subtraction operation is performed.
The above can be modified as follows

```
            uint256 _withdrawDifference;
            unchecked {
                _withdrawDifference = _taskCost - _newCost;
            }
```

File: Project.sol Line 616

```
        _costToAllocate -= _taskCost;
```

The above line cannot underflow due to the check on Line 614 which ensures that the above operation would only be performed if the value of `_costToAllocate` is greater than the value of `_taskCost`

File: Project.sol Line 663

```
        _costToAllocate -= _taskCost;
```

The above line cannot underflow due to the check on Line 661 which ensures that the above operation would only be performed if the value of `_costToAllocate` is greater than the value of `_taskCost`

File: Community.sol Line 794

```
        _lentAmount = _lentAndInterest - _repayAmount;
```

The above line cannot underflow due to the check on Line 792 which ensures that the above operation would only be performed if the value of `_lentAndInterest` is greater than the value of `_repayAmount`

File: Community.sol Line 798

```
        _interest -= _repayAmount;
```

The above line cannot underflow as it would only be evalauted if `_interest` is not less than `_repayAmount` . See Line 785

see resource

[G-09] Using unchecked blocks to save gas - Increments in for loop can be unchecked (save 30-40 gas per loop iteration)

The majority of Solidity for loops increment a uint256 variable that starts at 0. These increment operations never need to be checked for over/underflow because the variable will never reach the max number of uint256 (will run out of gas long before that happens). The default over/underflow check wastes gas in every iteration of virtually every for loop . eg.

e.g Let's work with a sample loop below.

```
for(uint256 i; i < 10; i++){
//doSomething
}
```

can be written as shown below.

```
for(uint256 i; i < 10;) {
  // loop logic
  unchecked { i++; }
}
```

We can also write it as an inlined function like below.

```
function inc(i) internal pure returns (uint256) {
  unchecked { return i + 1; }
}
for(uint256 i; i < 10; i = inc(i)) {
  // doSomething
}
```

Affected code

File: HomeFiProxy.sol line 87

```
        for (uint256 i = 0; i < _length; i++) {
            _generateProxy(allContractNames[i], _implementations
        }
```

The above should be modified to:

```
        for (uint256 i = 0; i < _length;) {
            _generateProxy(allContractNames[i], _implementations
                unchecked {
                        ++i;
                }
        }
```

Other Instances to modify

File: Project.sol Line 248

```
        for (uint256 i = 0; i < _length; i++) {
```

File: Project.sol Line 311

```
        for (uint256 i = 0; i < _length; i++) {
            _inviteSC(_taskList[i], _scList[i], false);
        }
```

File: Project.sol Line 322

```
        for (uint256 i = 0; i < _length; i++) {
```

File: Tasks.sol Line 181

```
            for (uint256 i = 0; i < _length; i++) _alerts[i] = _self
```

see resource

[G-10] Use Custom Errors instead of Revert Strings to save Gas

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met).
Custom errors save ~50 gas each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas

Custom errors are defined using the error statement, which can be used inside and outside of contracts (including interfaces and libraries).
see Source

File: DebtToken.sol line 31

```
            require(
                communityContract == _msgSender(),
                "DebtToken::!CommunityContract"
            );
```

File: DebtToken.sol line 50

```
            require(_communityContract != address(0), "DebtToken::0
```

File: DebtToken.sol line 96

```
            revert("DebtToken::blocked");
```

File: DebtToken.sol line 104

```
            revert("DebtToken::blocked");
```

File: ProjectFactory.sol line 36

```
            require(_address != address(0), "PF::0 address");
```

File: ProjectFactory.sol line 64

```
            require( _msgSender() == IHomeFi(homeFi).admin(), "Proje
```

File: ProjectFactory.sol line 84

```
            require(_msgSender() == homeFi, "PF::!HomeFiContract");
```

Note: see warden's **original submission** *for full list of instances.*

[G-11] x += y costs more gas than x = x + y for state variables

File: Project.sol Line 179

Project.sol.updateProjectHash() - (Saves ~19 gas)

Average gas before modification: 54538
Average gas after modification: 54519

```
            hashChangeNonce += 1;
```

The above should be modified to

```
            hashChangeNonce = hashChangeNonce + 1;
```

File: Project.sol Line 290

Project.sol.updateTaskHash() - (Saves ~19 gas)

Average gas before modification: 58185

Average gas after modification: 58166

```
        hashChangeNonce += 1;
```

File: HomeFi.sol Line 289

HomeFi.sol.mintNFT()

```
        projectCount += 1;
```

[G-12] Using bools for storage incurs overhead

```
    // Booleans are more expensive than uint256 or any type that
    // word because each write operation emits an extra SLOAD tc
    // slot's contents, replace the bits taken up by the boolear
    // back. This is the compiler's defense against contract upç
    // pointer aliasing, and it cannot be disabled.
```

See source

Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past.

Instances affected include

File: HomeFiProxy.sol line 30

```
    mapping(address => bool) internal contractsActive;
```

File: Disputes.sol Line 144

```
            bool _ratify
```

File: HomeFi.sol [Line 50](#)

```
        bool public override addrSet;
```

File: Project.sol [Line 68](#)

```
        bool public override contractorConfirmed;
```

File: Project.sol [Line 84](#)

```
        mapping(address => mapping(bytes32 => bool)) public override
```

File: Project.sol [Line 412](#)

```
            bool _unapproved = false;
```

File: Project.sol [Line 582](#)

```
            bool _exceedLimit;
```

[G-13] Using private rather than public for constants, saves gas

If needed, the value can be read from the verified contract source code. Savings are due to the compiler not having to create non-payable getter functions for deployment calldata, and not adding another entry to the method ID table.

File: Project.sol [Line 60](#)

```
    uint256 public constant override VERSION = 25000;
```

[G-14] Not using the named return variables when a function returns, wastes deployment gas

File: Project.sol Line 716-723

```
    function getAlerts(uint256 _taskID)
        public
        view
        override
        returns (bool[3] memory _alerts)
    {
        return tasks[_taskID].getAlerts();
    }
```

Dravee (warden) commented:

> Overall a high quality gas report IMHO. The warden starts with the most manual and interesting findings: storage reading optimizations. There are also the `unchecked` blocks. The gas savings are almost always mentioned too.

> **Analysis:**

> **[G-01] Cache storage values in memory to minimize SLOADs**

- HomeFi.sol.createProject(): projectCount should be cached(Saves ~ 71 gas)
- HomeFi.sol.mintNFT(): projectCount should be cached
- Project.sol.updateProjectHash(): hashChangeNonce should be cached (saves ~ 101 gas)
- Project.sol.updateTaskHash(): hashChangeNonce should be cached(saves ~ 98 gas)
- Project.sol.allocateFunds():_changeOrderedTask.length should be cached(Saves ~ 118 gas)
- Community.sol.createCommunity():communityCount should be cached(Saves ~186 gas)

> Valid

> **[G-02] Cache the length of arrays in loops**

- Project.sol.allocateFunds(): _changeOrderedTask.length should be cached - _changeOrderedTask is a storage array

> Valid

> **[G-03] ++i costs less gas compared to i++ or i += 1 in for loops (~5 gas per iteration)**

> Valid

> **[G-04] ++x is more efficient than x++(Saves ~6 gas)**

> Valid, kinda same as above (pre-increments)

> **[G-05] Splitting require() statements that use && saves gas - (saves 8 gas per &&)**

> Valid on Optimizer with 200 runs
> **[G-06] Comparisons: != is more efficient than in require (6 gas less)**

> Valid with Solidity 0.8.6 < 0.8.13

> **[G-07] Emitting storage values instead of the memory one(saves ~101 gas)**

> Valid

> **[G-08] Using unchecked blocks to save gas**

> Valid and well explained. I believe only 1 instance is missing in the solution:

- [Project.sol#L440]()

```
File: Project.sol
438:                    else if (totalLent - _totalAllocated >=
439:                        // Increase the difference of new c
```

```
438  440:                          totalAllocated += _newCost - _taskC
```

> **[G-09] Using unchecked blocks to save gas - Increments in for loop can be unchecked (save 30-40 gas per loop iteration)**

> Valid

> **[G-10] Use Custom Errors instead of Revert Strings to save Gas**

> Valid
> **[G-11] x += y costs more gas than x = x + y for state variables**

- Project.sol.updateProjectHash() - (Saves ~19 gas)
- Project.sol.updateTaskHash() - (Saves ~19 gas)
- HomeFi.sol.mintNFT()

> Valid, but could've saved more gas with `++x` instead of `x += 1`

> **[G-12] Using bools for storage incurs overhead**

> Valid but partially true as not all mentioned booleans are state booleans (some are memory ones or function arguments).

> **[G-13] Using private rather than public for constants, saves gas**

> I believe it's invalid here as this specific constant needs to be public.

> **[G-14] Not using the named return variables when a function returns, wastes deployment gas**

> From memory, this has actually been debunked (the optimizer takes care of it). So, invalid, but could be NC.

Jack the Pug (judge) commented:

> This is 💯!

parv3213 (Rigor) commented:

> [G-13] seems invalid, as the project version must be a public variable.

Jack the Pug (judge) commented:

> Re: [G-13], This depends on how the `VERSION()` method is going to be used. As it's inherited from `IProject`, it's probably required by the front-end, thus, I agree that this one is more likely to be invalid.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top