智能合约安全审计报告

NEST Protocol (V3)

SECBIT

Jul 12, 2020

安比（SECBIT）实验室致力于解决区块链全生态的安全问题，提供区块链全生态的安全服务。作为中国信息通信研究院区块链安全技术组成员，参与编写区块链安全白皮书和参与制定区块链安全审计规范。

安比（SECBIT）实验室智能合约审计从合约的技术实现、业务逻辑、接口规范、Gas 优化、发行风险等维度，由两组专业审计人员分别独立进行安全审计，审计过程借助于安比实验室研发的一系列形式化验证、语义分析等工具进行扫描检测，力求尽可能全面的解决所有安全问题。

1. 综述

NEST Protocol (V3)是一个分布式价格预言机网络协议。安比（SECBIT）实验室于 2020 年 4 月 28 日至 7 月 12 日对合约进行安全审计，审计过程从**代码漏洞**，**逻辑漏洞**和**发行风险评估**三个维度对代码进行分析。审计结果表明，NEST Protocol (V3) 并未包含致命的安全漏洞，安比（SECBIT）实验室给出了如下几点逻辑实现存疑和代码优化建议项（详见第4章节）。

风险类型	描述	风险级别	状态
代码实现	4.3.1 Nest_3_NestAbonus : reloadTimeAndMapping() 中 `_snapshot`未实现判断分红合约是否完成快照的功能	提示	已修复
代码实现	4.3.2 Nest_3_VoteFactory : changeStateOfEmergency() 中紧急状态修改的实现逻辑存疑	低	已修复
代码实现	4.3.3 Nest_3_VoteContract : constructor()中，当投票合约为紧急状态时，总票数与通过票数计算与实际不符	低	已修复
代码实现	4.3.4 Nest_3_OfferPrice与Nest_NToken_OfferPrice合约中，中`blockAddress`的更新与合约设计文档不一致	提示	已修复
代码实现	4.3.5 Nest_3_NestAbonus合约中，当预期分红增量比例变化时，会对预期分红的计算产生较大影响	低	已确认
代码优化	4.3.6 Nest_3_MiningContract： oreDrawing() 中，对于同一区块内的多个交易，存在冗余的计算	提示	已修复
代码规范	4.3.7 Nest_3_MiningContract 中，`_blockMining`从未初始化与更新	提示	已修复
代码实现	4.3.8 Nest_NToken_OfferMain合约运行四年后，挖矿功能失效	中	已修复
实现漏洞	4.3.9 Nest_3_OfferMain与Nest_NToken_OfferMain中，`turnOut()`函数存在重入攻击风险	低	已修复
代码优化	4.3.10 Nest_NToken_TokenAuction合约中，offer()函数可在一开始判断交易的ERC20 Token是否有对应的NToken	提示	已修复
代码实现	4.3.11 Nest_NToken_TokenAuction合约中，未经发起拍卖可进行上币操作	中	已修复

实现漏洞	4.3.12 NToken `transferFromForOfferPrice()` 权限校验存在问题	高	已修复
代码实现	4.3.13 部分涉及任意 token 转账的地方未考虑兼容性问题	低	已修复

2. 项目信息

该部分描述了项目的基本信息和代码组成。

2.1 基本信息

以下展示了 NEST Protocol (V3) 的基本信息：

- **项目网站**
  - https://nestprotocol.org/
- **项目白皮书**
  - https://nestprotocol.org/assets/pdf/zhnestwhitepaper.pdf
- **合约代码**
  - 由NEST Core团队提供

2.2 合约列表

以下展示了 Nest Protocol (V3) 项目包含的主要合约列表：

合约名称	描述
IBNEST	NEST Token合约
SuperMan	守护者节点Token合约
Nest_3_Abonus	ETH分红合约
Nest_3_Leveling	平准合约
Nest_3_NestAbonus	分红逻辑合约
Nest_3_NestSave	NEST锁仓合约
Nest_3_NTokenAbonus	nToken分红池合约
Nest_3_MiningContract	矿池合约
Nest_3_OfferMain	报价合约

Nest_3_OfferPrice	价格合约
Nest_3_OfferPriceAdmin	价格调用管理合约
NEST_NodeAssignment	守护者节点分配合约
NEST_NodeAssignmentData	守护者节点记录合约
NEST_NodeSave	守护者节点NEST存储合约
Nest_NToken_TokenAuction	nToken拍卖合约
Nest_NToken_TokenMapping	nToken映射合约
Nest_NToken_OfferMain	nToken报价合约
Nest_NToken_OfferPrice	nToken价格合约
Nest_3_VoteFactory	投票工厂合约

2.3 审计记录

1. 2020/04/28 开始审计
2. 2020/05/17 第一次交付审计问题列表
3. 2020/06/02 第二次交付审计问题列表
4. 2020/06/28 第三次交付审计问题列表
5. 2020/07/12 交付审计报告

审计期间，NEST Protocol V3 结合审计意见和自身需求，在设计与实现上经历多次改动。本审计报告是对整个审计过程的要点记录，部分内容可能无法与最新代码完全对应。

3. 代码分析

该部分描述了项目代码的详细内容分析，从「角色分类」和「功能分析」这两部分来进行说明。

3.1 角色分类

NEST Protocol (V3) 中主要涉及以下几种关键角色，分别是协议管理员（Protocol Owner）、守护者节点（NestNode）、报价者（Offerer）、NToken合约拥有者（NToken Owner）和 普通用户（Common Account）。

- 协议管理者（Protocol Owner）
  - 描述

    协议的拥有者
  - 功能权限
    - 协议中所有合约的管理员
    - 添加或者删除管理员地址
    - 设置协议合约映射地址
    - 设置协议合约的参数
    - 管理价格合约黑名单
  - 授权方式

    合约的创建者，被投票通过的账户或者由管理员授权
- 守护者节点（NestNode）
  - 描述

    NestNode Token的持有者，称作守护者节点
  - 功能权限
    - 对账户上的NestNode Token进行转账
    - 获得NEST预言机报价产出的NEST Token分红
    - 创建与参与NEST投票合约
    - 触发系统进入紧急状态
  - 授权方式

    NestNode Token的持有者
- 报价者(Offerer)
  - 描述

    提供价格的参与者，包含报价矿工和验证者。报价矿工：主动发起报价的账户，通过质押双向资产，为NEST预言机提供报价。验证者：与报价单达成交易，被强制跟随报价的账户。
  - 功能权限
    - 报价矿工可获取挖矿收益
    - 价格被调用预言机查询时获得收益
    - 报价单失效后取回报价单中剩余资产
  - 授权方式

    主动报价，或者成交报价时被强制跟随报价
- NToken合约拥有者（NToken Owner）
  - 描述

    NToken合约拥有者
  - 功能权限

        - NToken报价时获取挖矿收益
        - 转让NToken合约拥有者身份
    - 授权方式

      NToken拍卖成功者，或者通过合约拥有者转让授权

- 普通用户（Common Account）
    - 描述

      持有NEST Token的用户

    - 功能权限
        - 对账户上的NEST Token进行转账
        - 授权其他账户转账
        - 存入NEST Token以参与NEST投票合约的投票
        - 存入NEST Token参与分红
        - 发起与参与NToken拍卖合约
    - 授权方式

      NEST Token的持有者

3.2 功能分析

NEST Protocol (V3)合约是一个分布式价格预言机网络协议，合约中报价矿工通过主动发起报价，获取NEST Token与NToken的挖矿收益。NEST Token与NToken的挖矿收益分布在每一个区块中，每个报价矿工都可以瓜分所在区块的挖矿收益。每一次报价都需要抵押报价对应的两种资产，其他账户可以吃单，验证者以报价单价格兑换资产，吃单时需要重新报价，抵押资产并挂出新的报价单。价格偏离上一次报价较大时，验证者需要加大抵押的资产，从而减小发生恶意报价的风险。

报价单具有一定的生命周期，在报价单的生命周期内，被吃单会产生新的报价单，资产交易对的报价也同时更新，更接近于现实中的报价。NEST Protocol (V3)合约通过统计一定时间内挂出和成交的交易，对报价单对应的资产价格进行统计，形成链上价格预言机。

报价矿工通过报价挖矿获得NEST Token，持有NEST Token的用户可以选择将NEST Token锁仓，以定期获取分红以及参与投票。可以发起与参与NToken拍卖合约，拍卖成功者成为NToken合约拥有者，获得NToken挖矿的收益。

守护者节点持有NestNode Token，可以定期得到NEST Token分红。守护者节点可以发起投票，投票由NEST Token持有者与守护者节点参与，以决定合约是否升级。

NEST Protocol (V3)合约具有分红体系，激励NEST Token持有者锁仓得到定期分红。

合约实现关键功能分为以下几项：

- 报价矿工对NEST与NToken的挖矿

报价挖矿通过调用Nest_3_OfferMain与Nest_NToken_OfferPrice合约中的`offer()`函数，主动发起报价参与挖矿，`createOffer()`函数生成报价单，`oreDrawing()`与`mining()`函数用于挖矿Token数量的计算与分配。

- 验证者对NEST与NToken报价单的吃单操作

  NEST报价单与NToken报价单的吃单操作实现函数为Nest_3_OfferMain与Nest_NToken_OfferMain合约中的`sendEthBuyErc()`与`sendErcBuyEth()`函数，根据报价单中两种资产的报价，验证吃单并同时调用`createOffer()`函数抵押资产并生成新的报价单。

- 价格查询功能

  NEST Token持有者通过调用Nest_3_OfferPrice合约中的`updateAndCheckPriceNow()`与`updateAndCheckPriceList()`，同时转账NEST Token与NToken为查询价格付费。

4. 审计详情

该部分描述合约审计流程和详细结果，并对发现的问题（代码漏洞，代码规范和逻辑漏洞），合约发行的风险点和附加提示项进行详细的说明。

4.1 审计过程

本次审计工作，严格按照安比（SECBIT）实验室审计流程规范执行，从代码漏洞，逻辑问题以及合约发行风险三个维度进行全面分析。审计流程大致分为四个步骤:

- 各审计小组对代码进行逐行分析，根据审计内容要求进行审计
- 各审计小组对漏洞和风险进行评估
- 审计小组之间交换审计结果，并对审计结果进行逐一审查和确认
- 审计小组配合审计负责人生成审计报告

4.2 审计结果

本次审计首先经过安比（SECBIT）实验室推出的分析工具 adelaide、 sf-checker 和 badmsg.sender（内部版本）扫描，再利用开源安全分析工具 Mythril、Slither、SmartCheck 以及 Securify 检查，检查结果由审计小组成员详细确认。审计小组成员对合约源码和电路代码进行逐行检查、评估，汇总审计结果。审计内容总结为如下 21 大项。

编号	分类	结果

1	合约各功能能够正常执行	通过
2	合约代码不存在明显的漏洞（如整数溢出）	通过
3	能够通过编译器的编译并且编译器没有任何警告输出	通过
4	合约代码能够通过常见检测工具检测，并无明显漏洞	通过
5	不存在明显的 Gas 损耗	不通过
6	符合 EIP20 标准规范	通过
7	底层调用（call，delegatecall，callcode）或内联汇编的操作不存在安全隐患	通过
8	代码中不包含已过期或被废弃的用法	通过
9	代码实现清晰明确，函数可见性定义明确，变量数据类型定义明确，合约版本号明确	通过
10	不存在冗余代码	不通过
11	不存在受时间和外部网络环境影响的隐患	通过
12	业务逻辑实现清晰明确	通过
13	代码实现逻辑与注释，项目白皮书等资料保持一致	通过
14	代码不存在设计意图中未提及的逻辑	通过
15	业务逻辑实现不存在疑义	通过
16	不存在危及项目方利益的明确风险	通过
17	不存在危及相关机构如交易所，钱包，DAPP 方利益的明确风险	通过
18	不存在危及普通持币用户利益的明确风险	通过
19	不存在修改他人账户余额的特权	通过
20	不存在铸币权限	通过

21	多管理角色下，管理权限划分明确，各权限优先级划分明确		通过

4.3 问题列表

4.3.1 Nest_3_NestAbonus : reloadTimeAndMapping()中 _snapshot未实现判断分红合约是否完成快照的功能

风险类型	风险级别	影响点	状态
代码实现	提示	功能失效	已修复

问题描述

Nest_3_NestAbonus 中 reloadTimeAndMapping() 通过_snapshot映射判断分红合约是否进行过快照，然而_snapshot在快照后没有更新数值，因此每次判断都为false。

```
  if (_snapshot[token][_times.sub(1)] == false) {
              if (token == address(0x0)) {
                  _abonusMapping[address(0x0)] =
_abonusContract.getETHNum();
                //  快照 eth 数量
              } else {
                  _abonusMapping[address(token)] =
ERC20(address(token)).balanceOf(address(_NTokenAbonus));
              //  快照 token数量
              }
          }
```

修改建议

建议快照更新后将_snapshot[token][_times.sub(1)]赋值为true。

状态

已按照修改建议修复。

4.3.2 Nest_3_VoteFactory : changeStateOfEmergency() 中紧急状态修改的实现逻辑存疑

风险类型	风险级别	影响点	状态
代码实现	低	实现逻辑	已修复

问题描述

```
    function changeStateOfEmergency() public {
        if (_stateOfEmergency == false) {
            require(_NNToken.balanceOf(address(this)) ==
_emergencyNNAmount);
            _stateOfEmergency = true;
            _emergencyTime = now;
        } else {
            if (_NNToken.balanceOf(address(this)) ==
_emergencyNNAmount || now > _emergencyTime.add(_emergencyTimeLimit)) {
                _stateOfEmergency = false;
                _emergencyTime = 0;
            }
        }
    }
```

`Nest_3_VoteFactory`中紧急状态修改通过`changeStateOfEmergency()`实现，满足`_NNToken.balanceOf(address(this)) == _emergencyNNAmount`条件时，合约的紧急状态可由任意用户进行切换，且当存入合约的NestNode Token数量大于`_emergencyNNAmount`时，不满足紧急状态修改的条件。

修改建议

建议判断调用函数的用户是否为超级节点，即增加`require(_emergencyPerson[address(tx.origin)] > 0);`，存入合约的NestNode Token数量的判断条件改为`_NNToken.balanceOf(address(this)) >= _emergencyNNAmount`。

状态

已按照修改建议修复。

4.3.3 Nest_3_VoteContract : constructor()中，当投票合约为紧急状态时，总票数与通过票数计算与实际不符

风险类型	风险级别	影响点	状态
代码实现	低	在有限范围内影响成交价格	已修复

问题描述

投票合约在紧急状态下，超级节点不参与投票，_totalAmount被设置初值，默认超级节点都进行了投票，_circulation表示当前投票通过的票数。在_totalAmount与_circulation的计算中，_nestAbonus.checkAllValueMapping(2)即表示Nest Token流通量的快照，由Nest_3_NestAbonus:allValue()计算得到。因此紧急状态下，_circulation与_totalAmount的计算与实际不符。

```
        constructor (address contractAddress) public {
        //  初始化
        _voteFactory = Nest_3_VoteFactory(address(msg.sender));
        _nestToken = ERC20(_voteFactory.checkAddress("nest"));
        _NNToken = ERC20(_voteFactory.checkAddress("nestNode"));
        _implementContract =
Nest_3_Implement(address(contractAddress));
        _implementAddress = address(contractAddress);
        _destructionAddress =
address(_voteFactory.checkAddress("nest.v3.destruction"));
        _creator = address(tx.origin);
        _NNUsedCreate = _voteFactory.checkNNUsedCreate();
        _createTime = now;

        _endTime = _createTime.add(_voteFactory.checkLimitTime());
        if (_voteFactory.checkStateOfEmergency() == false) {
            _miningSave =
Nest_3_MiningSave(_voteFactory.checkAddress("nest.v3.miningSave"));
            _nestSave =
Nest_3_NestSave(_voteFactory.checkAddress("nest.v3.nestSave"));
            _circulation = (uint256(10000000000
ether).sub(_nestToken.balanceOf(address(_miningSave))).sub(_nestToken.
balanceOf(address(_destructionAddress)))).mul(_voteFactory.checkCircul
ationProportion()).div(100);
            uint256 NNProportion = _voteFactory.checkNNProportion();
            _NNSingleVote = (uint256(10000000000
ether).sub(_nestToken.balanceOf(address(_miningSave)))).mul(NNProporti
on).div(150000);
        } else {
            _nestAbonus =
Nest_3_NestAbonus(_voteFactory.checkAddress("nest.v3.nestAbonus"));
            _circulation = (uint256(10000000000
ether).sub(_nestAbonus.checkAllValueMapping(2)).sub(_nestToken.balance
Of(address(_destructionAddress)))).mul(_voteFactory.checkCirculationPr
oportion()).div(100);
            uint256 NNProportion = _voteFactory.checkNNProportion();
            _totalAmount = (uint256(10000000000
ether).sub(_nestAbonus.checkAllValueMapping(2))).mul(NNProportion).div
(100);
        }
    }
```

```
function allValue() public view returns (uint256) {
    uint256 all = 10000000000 ether;
    uint256 leftNum =
all.sub(_nestContract.balanceOf(address(_voteFactory.checkAddress("nes
t.v3.miningSave")))).sub(_nestContract.balanceOf(address(_destructionA
ddress)));
    return leftNum;
}
```

修改建议

建议紧急状态下_circulation与 _totalAmount的计算改为_circulation = _nestAbonus.checkAllValueMapping(2).mul(_voteFactory.checkCirculationProportion()).div(100)与_totalAmount = _nestAbonus.checkAllValueMapping(2).add(_nestToken.balanceOf(address(_destructionAddress))).mul(NNProportion).div(100)

状态

紧急状态下的投票逻辑已修改，问题已修复。

4.3.4 Nest_3_OfferPrice与Nest_NToken_OfferPrice合约中，中blockAddress的更新与合约设计文档不一致

风险类型	风险级别	影响点	状态
代码实现	提示	文档和实现不一致	已更新

问题描述

合约设计文档中blockAddress用于记录区块内第一个报价的矿工，addPrice()函数的代码实现中blockAddress记录的是区块中最后一个报价的矿工。

```
    function addPrice(uint256 ethAmount, uint256 tokenAmount, uint256
endBlock, address tokenAddress) public onlyFactory {
    ......
        //  更新最后一个报价矿工
        blockAddress[block.number][tokenAddress] = address(tx.origin);
    }
```

修改建议

确定blockAddress的含义，修改代码或者设计文档。

状态

已确定blockAddress表示最后一个报价的矿工。

4.3.5 Nest_3_NestAbonus合约中，当预期分红增量比例变化时，会对预期分红的计算产生较大影响。

风险类型	风险级别	影响点	状态
代码实现	低	实现逻辑	已确认

问题描述

在NEST3.0经济模型中，NEST流通量每增加一亿，预期最低分红的ETH数量增加一定的比例，其数量变化应该是连续的。预期最低分红在levelingResult()函数中计算，分红比例变化从NEST流通量为0时开始计算，即

$$(1 + expectedIncrement)^{nestAllValue/100000000}$$

因此，当预期分红增量比例变化时，当期最低分红的ETH数量的计算会产生指数性的影响。

```
uint256 miningAmount = allValue().div(100000000 ether);
        uint256 minimumAbonus = _expectedMinimum;
        for (uint256 i = 0; i < miningAmount; i++) {
            minimumAbonus =
minimumAbonus.add(minimumAbonus.mul(_expectedIncrement).div(100));
        }
```

修改建议

建议修改预期最低分红的计算方法（如逐次更新存储），减少参数变化对当期最低分红计算产生的影响。

状态

已确认保持现有逻辑。

4.3.6 Nest_3_MiningContract： oreDrawing() 中，对于同一区块内的多个交易，存在冗余的计算。

风险类型	风险级别	影响点	状态
代码优化	提示	冗余代码消耗不必要的gas	已修复

问题描述

NEST协议中，用户在报价的同时进行挖矿，同一区块内的第一个报价的交易可以从矿池合约中挖到NEST Token。挖矿数量通过changeBlockAmountList()函数计算，区块内第一次调用返回挖矿数量，_latestMining被更新为block.number，其后同一区块内调用该函数时，返回值是0。因此当挖矿数量为0时，oreDrawing()函数中的_nestContract.transfer(address(msg.sender), miningAmount);与emit OreDrawingLog(block.number,miningAmount,msg.value);都是冗余操作。

```
    function oreDrawing() public payable returns (uint256) {
        require(address(msg.sender) == _offerFactoryAddress, "No
authority");
        //  更新出矿量列表
        uint256 miningAmount = changeBlockAmountList();
        //  转手续费
        repayEth(msg.value);
        //  转 nest
        _nestContract.transfer(address(msg.sender), miningAmount);

        emit OreDrawingLog(block.number,miningAmount,msg.value);
        return miningAmount;
    }
```

修改建议

建议在oreDrawing()的实现中，先通过_latestMining 是否等于 block.number判断挖矿数量是否为0，再进行相关代码的执行。

状态

已按照修改建议修复。

4.3.7 Nest_3_MiningContract 中，_blockMining从未初始化与更新。

风险类型	风险级别	影响点	状态
代码规范	提示	存在未使用的映射	已修复

问题描述

Nest_3_MiningContract 中，_blockMining用于存储区块号对应的出矿量，此映射未在代码中进行更新。

修改建议

建议在oreDrawing()的实现中，有效挖矿后更新_blockMining。

状态

_blockMining变量被删除，问题已修复。

4.3.8 Nest_NToken_OfferMain合约运行四年后，挖矿功能失效。

风险类型	风险级别	影响点	状态
代码实现	中	NToken挖矿功能失效	已修复

问题描述

Nest_NToken_OfferMain 中，_attenuationAmount数组用于存储不同时间阶段的挖矿数量，数组长度为4。因此当合约运行到第五年，oreDrawing()函数在访问_attenuationAmount[4]时数组越界，导致函数执行出错，NToken挖矿功能失效。

```
    function oreDrawing(address NToken) private returns(uint256) {
        Nest_NToken miningToken = Nest_NToken(NToken);
        (uint256 createBlock, uint256 recentlyUsedBlock) =
miningToken.checkBlockInfo();
        uint256 attenuationPointOld =
recentlyUsedBlock.sub(createBlock).div(_blockAttenuation);
        uint256 attenuationPointNow =
block.number.sub(createBlock).div(_blockAttenuation);
        uint256 miningAmount;
        uint256 attenuationPointend =
createBlock.add(attenuationPointOld.add(1).mul(_blockAttenuation));
            for (uint256 i = attenuationPointOld; i <
attenuationPointNow; i++) {
                miningAmount =
miningAmount.add(attenuationPointend.sub(recentlyUsedBlock).mul(_atten
uationAmount[i]));
                recentlyUsedBlock = attenuationPointend;
                attenuationPointend =
attenuationPointend.add(_blockAttenuation);
            }
        miningAmount =
miningAmount.add(_attenuationAmount[attenuationPointNow].mul(block.num
ber.sub(recentlyUsedBlock)));
        miningToken.increaseTotal(miningAmount);
        emit OreDrawingLog(block.number, miningAmount, NToken);

        return miningAmount;
    }
```

修改建议

建议在oreDrawing()的实现中，增加对数组下标的判断，下标大于3时返回_attenuationAmount[3]的值。

状态

已修复。

4.3.9 Nest_3_OfferMain与Nest_NToken_OfferMain中，`turnOut()`函数存在重入攻击风险。

风险类型	风险级别	影响点	状态
实现漏洞	低	违反安全开发规范	已修复

问题描述

Nest_3_OfferMain与Nest_NToken_OfferMain中，turnOut()函数在执行repayEth(offerPriceData.owner, offerPriceData.ethAmount)时，若offerPriceData.owner为合约账户，该合约收到转账后在回调函数中重新调用turnOut()函数即可发起重入攻击。

虽然offer()函数限制报价发起人不可为合约账户，但是sendEthBuyErc()与sendErcBuyEth()未对交易发起人进行限制，可在函数中调用createOffer()发起报价。因此合约账户通过调用这两个函数发起报价，在报价失效后即可发起重入攻击。

由于repayEth()函数中使用了transfer()进行以太币转账，调用固定给2300的Gas，因此目前无法造成实质危害。

但ERC20(address(offerPriceData.tokenAddress)).safeTransfer()的具体实现未知且不可预测，因此仍存在一定风险。建议此处遵循智能合约安全开发规范，采用Checks-Effects-Interactions 的范式，先进行校验，再对关键参数进行修改，最后进行外部交互。

```
    function turnOut(address contractAddress) public {
        uint256 index = toIndex(contractAddress);
        Nest_3_OfferPriceData storage offerPriceData = _prices[index];
        require(checkContractState(offerPriceData.blockNum) == 1,
"Offer status error");

        // 取出ETH
        if (offerPriceData.ethAmount > 0) {
            repayEth(offerPriceData.owner, offerPriceData.ethAmount);
            offerPriceData.ethAmount = 0;
        }
```

```
        // 取出ERC20
        if (offerPriceData.tokenAmount > 0) {

 ERC20(address(offerPriceData.tokenAddress)).safeTransfer(offerPriceDa
ta.owner, offerPriceData.tokenAmount);
            offerPriceData.tokenAmount = 0;
        }
        // 挖矿结算
        if (offerPriceData.serviceCharge > 0) {
            uint256 myMiningAmount =
offerPriceData.serviceCharge.mul(_offerBlockMining[offerPriceData.bloc
kNum]).div(_offerBlockEth[offerPriceData.blockNum]);
            _nestToken.safeTransfer(offerPriceData.owner,
myMiningAmount);
            offerPriceData.serviceCharge = 0;
        }

    }
```

```
    function turnOut(address contractAddress) public {
        uint256 index = toIndex(contractAddress);
        Nest_NToken_OfferPriceData storage offerPriceData =
_prices[index];
        require(checkContractState(offerPriceData.blockNum) == 1,
"Offer status error");
        // 取出ETH
        if (offerPriceData.ethAmount > 0) {
            repayEth(offerPriceData.owner, offerPriceData.ethAmount);
            offerPriceData.ethAmount = 0;
        }
        // 取出ERC20
        if (offerPriceData.tokenAmount > 0) {

 ERC20(address(offerPriceData.tokenAddress)).transfer(offerPriceData.o
wner, offerPriceData.tokenAmount);
            offerPriceData.tokenAmount = 0;
        }
        // 挖矿结算
        if (offerPriceData.mining) {
            mining(offerPriceData.blockNum,
offerPriceData.tokenAddress);
            offerPriceData.mining = false;
        }
    }
```

修改建议

在sendEthBuyErc()与sendErcBuyEth()对交易发起人进行限制，并在调用repayEth()前执行offerPriceData.ethAmount = 0。也可考虑使用ReentrancyGuard 对所有必要函数进行重入保护。

状态

已修复。

4.3.10 Nest_NToken_OfferMain合约中，offer()函数可在一开始判断交易的ERC20 Token是否有对应的NToken。

风险类型	风险级别	影响点	状态
代码优化	提示	函数实现逻辑不合理	已修复

问题描述

Nest_NToken_OfferMain合约中，offer()函数在执行价格对比、报价创建、Token转账等操作后，才对NToken有效性进行判断。

修改建议

建议在offer()函数开头处增加对erc20Address对NToken映射的判断。

状态

已修复。

4.3.11 Nest_NToken_TokenAuction合约中，未经发起拍卖可进行上币操作。

风险类型	风险级别	影响点	状态
实现漏洞	中	NToken创建机制失效	已修复

问题描述

Nest_NToken_TokenAuction合约中，由于_auctionList[token]结构体未初始化时值为0，now > _auctionList[token].endTime与_nestToken.transfer(_destructionAddress, _auctionList[token].auctionValue)条件都可满足，即auctionSuccess()函数在拍卖未发起时可以正常执行，从而存储无效的NToken映射。用户无需Nest Token即可成功拍卖创建NToken合约，导致NToken创建机制失效。需要管理员在Nest_NToken_TokenMapping合约中重置NToken映射才可恢复正常。

```
function auctionSuccess(address token) public {
        Nest_3_NestAbonus nestAbonus =
Nest_3_NestAbonus(_voteFactory.checkAddress("nest.v3.nestAbonus"));
        uint256 nowTime = now;
        uint256 nextTime = nestAbonus.getNextTime();
        uint256 timeLimit = nestAbonus.checkTimeLimit();
        uint256 getAbonusTimeLimit =
nestAbonus.checkGetAbonusTimeLimit();
        require(!(nowTime >= nextTime.sub(timeLimit) && nowTime <=
nextTime.sub(timeLimit).add(getAbonusTimeLimit)), "Not time to
auctionSuccess");
        require(now > _auctionList[token].endTime, "Token is on
sale");
        //  初始化NToken
        Nest_NToken NToken = new Nest_NToken(strConcat("NToken",
getAddressStr(token)), strConcat("N", getAddressStr(token)),
address(_voteFactory), _createValue, _NTokenAbonusValue,
_NTokenAbonus, address(_auctionList[token].latestAddress));
        //  拍卖资金销毁
        require(_nestToken.transfer(_destructionAddress,
_auctionList[token].auctionValue), "Transfer failure");
        //  加入 NToken映射
        _tokenMapping.addTokenMapping(token, address(NToken));
    }
```

修改建议

建议在`auctionSuccess()`函数中增加对拍卖发起状态的判断。

状态

已修复。

4.3.12 NToken `transferFromForOfferPrice()` 权限校验存在问题

风险类型	风险级别	影响点	状态
实现漏洞	高	权限校验失效	已修复

问题描述

```
function transferFromForOfferPrice(address from, address to, uint256
value) public returns (bool) {
    _transfer(from, to, value);
    emit Approval(from, msg.sender, _allowed[from][msg.sender]);
    return true;
}
```

修改建议

移除该函数或添加合适的权限校验。

状态

已修复，已移除该函数。

4.3.13 部分涉及任意 token 转账的地方未考虑兼容性问题

风险类型	风险级别	影响点	状态
代码实现	低	token转账兼容性	已修复

问题描述

```
function startAnAuction(address token, uint256 auctionAmount) public {
    require(_tokenMapping.checkTokenMapping(token) == address(0),
"Token already exists");
    require(_auctionList[token].endTime == 0, "Token is on sale");
    require(auctionAmount >= _minimumNest, "AuctionAmount less than
the minimum auction amount");
    require(_nestToken.transferFrom(address(msg.sender),
address(this), auctionAmount), "Authorization failed");
    require(_tokenBlackList[token] == false);
    AuctionInfo memory thisAuction = AuctionInfo(now.add(_duration),
auctionAmount, address(msg.sender), auctionAmount);
    _auctionList[token] = thisAuction;
    _allAuction.push(token);
}
```

由于部分 token 的接口实现并未严格按照 ERC20 接口定义，与这类 token 交互时可能存在兼容性问题，进而影响协议正常运行。

修改建议

使用 `safeTransfer()` 和 `safeTransferFrom()` 函数处理 token 调用。

状态

已修复。

4.4 风险提示

4.4.1 NEST协议分红池规模较小时，分红机制可能因为交易gas消耗大于收益而失效。

风险类型	风险级别	影响点	状态
协议设计	提示	分红机制的稳定性	已确认

问题描述

NEST协议通过分红机制激励用户将NEST Token锁仓，定期领取分红。有锁仓的用户通过调用`Nest_3_NestAbonus:getAbonus()`领取分红奖励，此函数的执行中存在大量的更新与计算操作，如`levelingResult()`中对`minimumAbonus`的计算，增加用户发起交易所需的gas数量。由测试网部署和合约交易来看，调用一次`getAbonus()`需要消耗约16w的gas，以目前以太坊主网30GWei的gas price进行计算，则执行一次分红领取操作消耗 30GWei*160000≈0.005 Ether，因此用户收益达到0.005 Ether时，分红机制才是有效的。

```
    function levelingResult() private {
        uint256 thisAbonus = _abonusContract.getETHNum();
        if (thisAbonus > 10000 ether) {

 _abonusContract.getETH(thisAbonus.mul(_savingLevelThree).div(100),
address(_nestLeveling));
        } else if (thisAbonus > 1000 ether) {

 _abonusContract.getETH(thisAbonus.mul(_savingLevelTwo).div(100),
address(_nestLeveling));
        } else if (thisAbonus > 100 ether) {

 _abonusContract.getETH(thisAbonus.mul(_savingLevelOne).div(100),
address(_nestLeveling));
        }

        uint256 miningAmount = allValue().div(100000000 ether);
        uint256 minimumAbonus = _expectedMinimum;
        for (uint256 i = 0; i < miningAmount; i++) {
            minimumAbonus =
minimumAbonus.add(minimumAbonus.mul(_expectedIncrement).div(100));
        }
        uint256 nowEth = _abonusContract.getETHNum();
        if (nowEth < minimumAbonus) {
```

```
            _nestLeveling.tranEth(minimumAbonus.sub(nowEth),
address(_abonusContract));
        }
    }
```

修改建议

建议减少提取分红操作中不必要的代码执行，减少gas消耗。加大ETH分红池的规模，增加用户分红收益，提升用户参与分红的积极性。

状态

已确认保持现有逻辑。

4.4.2 NEST协议管理员与社区治理

风险类型	风险级别	影响点	状态
协议设计	提示	治理安全	已讨论

问题描述

NEST Protocol 设计与实现中存在部分 `onlyOwner` 的管理员操作接口，用于更改协议设置和进行协议更新。需确保管理员私钥安全从而保障协议不被恶意修改。

此外，为了规避管理员账户单点失效的风险，NEST Protocol 计划去除管理员权限，并转型为社区投票治理模式。投票治理目前灵活度较大，长期的社区治理安全则需要依赖“守护者节点”和投票参与者一起确保每次投票合约的实现安全。

状态

已讨论。

5. 结论

安比（SECBIT）实验室在对 NEST Protocol(V3) 合约进行分析后，发现部分安全问题和可优化项，并提出了对应的修复及优化建议，上文均已给出具体的分析说明。最终，本报告列出的问题，NEST Protocol 开发团队在最新版代码中均进行了修复。安比（SECBIT）实验室认为 NEST 协议通过经济激励与博弈机制实现了链上价格预言机，协议整体设计较为巧妙，充分调动了协议中各类角色的积极性，并特别从链上价格安全角度做了许多独特的设计和细节优化。

免责声明

SECBIT 智能合约安全审计从合约代码质量、合约逻辑设计和合约发行风险等方面对合约的正确性、安全性、可执行性进行审计，但不做任何和代码的适用性、商业模式和管理制度的适用性及其他与合约适用性相关的承诺。本报告为技术信息文件，不作为投资指导，也不为代币交易背书。

附录

漏洞风险级别介绍

风险级别	风险描述
高	可以严重损害合约完整性的缺陷，能够允许攻击者盗取以太币及Token，或者把以太币锁死在合约里等缺陷。
中	在一定限制条件下能够损害合约安全的缺陷，造成某些参与方利益损失的缺陷。
低	并未对合约安全造成实质损害的缺陷。
提示	不会带来直接的风险，但与合约安全实践或合约合理性建议有关的信息。

安比（SECBIT）实验室致力于参与共建共识、可信、有序的区块链经济体。

https://secbit.io

info@secbit.io

@secbit_io