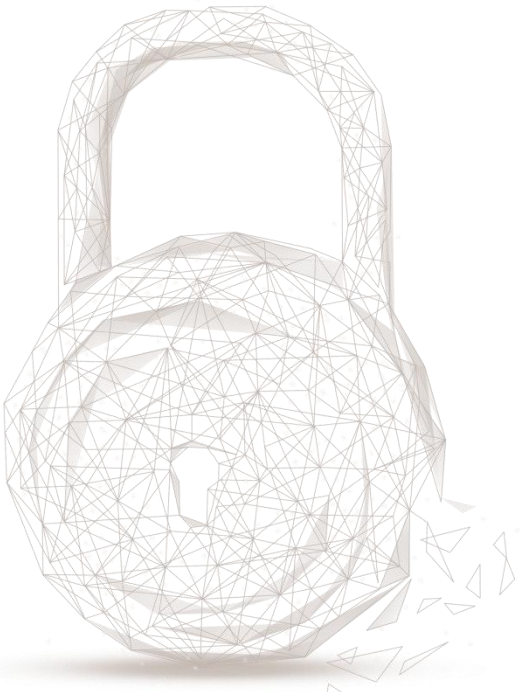
成都链安
BEOSIN

智能合约安全审计报告



成都链安 BEOSIN

审计编号：202108021900

审计项目名称：launchpad

报告查询名称：launchpad

审计项目链接：

https://github.com/ImpossibleFinance/launchpad-contracts

commit：cfd6d48428d266795ef8ef73e8aa0ddd08a7f354

合约审计开始日期：2021.07.29

合约审计完成日期：2021.08.02

审计结果：通过

审计团队：成都链安科技有限公司

审计类型及结果：

序号	审计类型	审计子项	审计结果
1	代码规范审计	编译器版本安全审计	通过
		弃用项审计	通过
		冗余代码审计	通过
		SafeMath 函数审计	通过
		require/assert 使用审计	通过
		gas 消耗审计	通过
		可见性规范审计	通过
		fallback 函数使用审计	通过
2	通用漏洞审计	整型溢出审计	通过
		重入攻击审计	通过
		伪随机数生成审计	通过
		交易顺序依赖审计	通过
		拒绝服务攻击审计	通过
		函数调用权限审计	通过
		call/delegatecall 安全审计	通过
		返回值安全审计	通过
		tx.origin 使用安全审计	通过
		重放攻击审计	通过
		变量覆盖审计	通过
3	业务审计	业务逻辑审计	通过
		业务实现审计	通过

于合约提供者在本报告出具前已向成都链安科技提供的文件和资料，且该部分文件和资料不存在任何缺失，被篡改，删减或隐瞒的前提下作出的；如提供的文件和资料存在信息缺失，被篡改，删减，隐瞒或反映的情况与实际情况不符等情况或提供文件和资料在本报告出具后发生任何变动的，成都链安科技对由此而导致的损失和不利影响不承担任何责任。成都链安科技出具的本审计报告系根据合约提供者提供的文件和资料依靠成都链安科技现掌握的技术而作出的，由于任何机构均存在技术的局限性，成都链安科技作出的本审计报告仍存在无法完整检测出全部风险的可能性，成都链安科技对由此产生的损失不承担任何责任。

本声明最终解释权归成都链安科技所有。

审计结果说明：

本公司采用形式化验证、静态分析、动态分析、典型案例测试和人工审核的方式对launchpad项目智能合约代码规范性、安全性以及业务逻辑三个方面进行多维度全面的安全审计。**经审计，launchpad项目智能合约通过所有检测项，合约审计结果为通过。**以下为本项目详细审计信息：

一、代码规范审计

1.1 编译器版本安全审计

老版本的编译器可能会导致各种已知的安全问题，建议开发者在代码中指定合约代码采用最新的编译器版本，并消除编译器告警。

本项目的所有合约指定了编译器版本为0.8.4以上，使用对应版本的编译器编译合约无任何编译器警告。

- **安全建议：无**
- **审计结果：通过**

1.2 弃用项审计

Solidity智能合约开发语言处于快速迭代中，部分关键字已被新版本的编译器弃用，如throw、years等，为了消除其可能导致的隐患，合约开发者不应该使用当前编译器版本已弃用的关键字。

- **安全建议：无**
- **审计结果：通过**

1.3 冗余代码审计

智能合约中的冗余代码会降低代码可读性，并可能需要消耗更多的gas用于合约部署，建议消除冗余代码。

- **安全建议：无**
- **审计结果：通过**

1.4 SafeMath功能审计

检查合约中是否正确使用SafeMath库内的函数进行数学运算，或者进行其他防溢出的检查。

- **安全建议：无**
- **审计结果：通过**

1.5 require/assert使用审计

Solidity使用状态恢复异常来处理错误。这种机制将会撤消对当前调用(及其所有子调用)中的状态所做的所有更改，并向调用者标记错误。函数assert和require可用于检查条件并在条件不满足时抛出异常。assert函数只能用于测试内部错误，并检查非变量。require函数用于确认条件有效性，例如输入变量，或合约状态变量是否满足条件，或验证外部合约调用的返回值。

- **安全建议：无**
- **审计结果：通过**

1.6 gas消耗审计

BSC虚拟机执行合约代码需要消耗gas，当gas不足时，代码执行会抛出out of gas异常，并撤销所有状态变更。合约开发者需要控制代码的gas消耗，避免因为gas不足导致函数执行一直失败。

本项目的IFAllocationMaster合约的getUserStakeWeight函数中，会遍历用户与全局记录的滞后记录数据，虽然getUserStakeWeight函数本身是view类型，但在IFAllocationSale合约中_purchase函数(非view)中也调用了此函数，因此，如果这里的滞后量过大，是有可能导致purchase函数因gas过高而调用失败。因此，建议管理员控制bumpSaleCounter函数调用的频率，避免用户因长期不操作而引发的滞后数量过大，而导致用户调用_purchase函数失败。

```
331     for (uint24 i = 0; i < numFinishedSalesDelta; i++) {
332       // get number of blocks passed at the current sale number
333       uint80 elapsedBlocks =
334         trackFinishedSaleBlocks[trackId][
335           closestUserCheckpoint.numFinishedSales + i
336         ] - currBlock;
337
338       // update stake weight
339       stakeWeight =
340         stakeWeight +
341         (uint104(elapsedBlocks) *
342           track.weightAccrualRate *
343           closestUserCheckpoint.staked) /
344         10**18;
345
346       // get amount of stake weight actively rolled over for this sale number
347       uint104 activeRolloverWeight =
348         trackActiveRollOvers[trackId][user][
349           closestUserCheckpoint.numFinishedSales + i
350         ];
```

图 1 deposit函数源码截图

- **安全建议：无**

- **审计结果：通过**

1.7 可见性规范审计

检查合约函数的可见性是否符合设计要求。

- **安全建议：无**
- **审计结果：通过**

1.8 fallback函数使用审计

检查在当前合约中是否正确使用了fallback函数。

- **安全建议：无**
- **审计结果：通过**

二、通用漏洞审计

2.1 整型溢出审计

整型溢出是很多语言都存在的安全问题，它们在智能合约中尤其危险。Solidity最多能处理256位的数字(2**256-1)，最大数字增加1会溢出得到0。同样，当数字为uint类型时，0减去1会下溢得到最大数字值。溢出情况会导致不正确的结果，特别是如果其可能的结果未被预期，可能会影响程序的可靠性和安全性。

本项目的合约使用了0.8.4以上版本的编译器，其内置了溢出检查的功能。如果对应的数学运算出现溢出时，会自动revert。

- **安全建议：无**
- **审计结果：通过**

2.2 重入攻击审计

重入漏洞是最典型的智能合约漏洞，该漏洞原因是Solidity中的call.value()函数在被用来发送BNB的时候会消耗它接收到的所有gas，当调用call.value()函数发送BNB的逻辑顺序存在错误时，就会存在重入攻击的风险。

本项目的合约继承了ReentrancyGuard合约，并且项目中关键函数均被nonReentrant修饰器所修饰，可很好的避免合约重入。

```
7  import '@openzeppelin/contracts/security/ReentrancyGuard.sol';
8  import 'hardhat/console.sol';
9
10 // IFAllocationMaster is responsible for persisting all launchpad state between project token sales
11 // in order for the sales to have clean, self-enclosed, one-time-use states.
12
13 // IFAllocationMaster is the master of allocations. He can remember everything and he is a smart guy.
14 contract IFAllocationMaster is Ownable, ReentrancyGuard {
15     using SafeERC20 for ERC20;
```

图 2 合约声明

- **安全建议：无**
- **审计结果：通过**

2.3 伪随机数生成审计

智能合约中可能会使用到随机数，在solidity下常见的是用block区块信息作为随机因子生成，但是这样使用是不安全的，区块信息是可以被矿工控制或被攻击者在交易时获取到，这类随机数在一定程度上是可预测或可碰撞的，比较典型的例子就是fomo3d的airdrop随机数可以被碰撞。

- **安全建议：无**
- **审计结果：通过**

2.4 交易顺序依赖审计

在BSC的交易打包执行过程中，面对相同难度的交易时，矿工往往会选择gas费用高的优先打包，因此用户可以指定更高的gas费用，使自己的交易优先被打包执行。

- **安全建议：无**
- **审计结果：通过**

2.5 拒绝服务攻击审计

拒绝服务攻击，即Denial of Service，可以使目标无法提供正常的服务。在以太坊智能合约中也会存在此类问题，由于智能合约的不可更改性，该类攻击可能使得合约永远无法恢复正常工作状态。导致智能合约拒绝服务的原因有很多种，包括在作为交易接收方时的恶意revert、代码设计缺陷导致gas耗尽等等。

- **安全建议：无**
- **审计结果：通过**

2.6 函数调用权限审计

智能合约如果存在高权限功能，如：铸币、自毁、change owner等，需要对函数调用做权限限制，避免权限泄露导致的安全问题。

- **安全建议：无**
- **审计结果：通过**

2.7 call/delegatecall安全审计

Solidity中提供了call/delegatecall函数来进行函数调用，如果使用不当，会造成call注入漏洞，例如call的参数如果可控，则可以控制本合约进行越权操作或调用其他合约的危险函数。

- **安全建议：无**
- **审计结果：通过**

2.8 返回值安全审计

在Solidity中存在transfer()、send()、call.value()等方法中，transfer转账失败交易会回滚，而send和call.value转账失败会return false，如果未对返回做正确判断，则可能会执行到未预期的逻辑；另外在BEP-20 Token的transfer/transferFrom功能实现中，也要避免转账失败return false的情况，以免造成假充值漏洞。

- **安全建议：无**
- **审计结果：通过**

2.9 tx.origin使用安全审计

在以太坊智能合约的复杂调用中，tx.origin表示交易的初始创建者地址，如果使用tx.origin进行权限判断，可能会出现错误；另外，如果合约需要判断调用方是否为合约地址时则需要使用tx.origin，不能使用extcodesize。

- **安全建议：无**
- **审计结果：通过**

2.10 重放攻击审计

重放攻击是指如果两份合约使用了相同的代码实现，并且身份鉴权在传参中，当用户在向一份合约中执行一笔交易，交易信息可以被复制并且向另一份合约重放执行该笔交易。

- **安全建议：无**
- **审计结果：通过**

2.11 变量覆盖审计

以太坊合约中存在着复杂的变量类型，例如结构体、动态数组等，如果使用不当，对其赋值后，可能导致覆盖已有状态变量的值，造成合约执行逻辑异常。

- **安全建议：无**
- **审计结果：通过**

三、业务审计

3.1 TestToken合约

因为本合约仅仅是用于测试代币，其实现均是引用于较为安全的openzeppelin库内合约，这里将不再详细讨论。

3.2 IFAllocationMaster合约

3.2.1 合约所有权管理

- **业务描述：**合约的管理员(默认为合约部署者)可调用本合约的 transferOwnership 函数转移合约所有权至指定地址；调用 renounceOwnership 函数放弃合约所有权。

- **相关函数：**transferOwnership、renounceOwnership
- **安全建议：无**
- **审计结果：通过**

3.2.2 添加track

- **业务描述：**合约所有者可调用addTrack函数添加一个新的track。如下图所示，由于该合约仅可由owner调用，并没有检查对应的参数，直接添加至对应数组，然后调用addTrackCheckpoint函数添加对应的快照信息。

```
126   // adds a new track
127   function addTrack(
128     string calldata name,
129     ERC20 stakeToken,
130     uint24 _weightAccrualRate,
131     uint64 _passiveRolloverRate,
132     uint64 _activeRolloverRate,
133     uint104 _maxTotalStake
134   ) external onlyOwner {
135     require(_weightAccrualRate != 0, 'accrual rate is 0');
136
137     // add track
138     tracks.push(
139       TrackInfo({
140         name: name, // name of track
141         stakeToken: stakeToken, // token to stake (e.g., IDIA)
142         weightAccrualRate: _weightAccrualRate, // rate of stake weight accrual
143         passiveRolloverRate: _passiveRolloverRate, // passive rollover
144         activeRolloverRate: _activeRolloverRate, // active rollover
145         maxTotalStake: _maxTotalStake // max total stake
146       })
147     );
148
149     // add first track checkpoint
150     addTrackCheckpoint(
151       uint24(tracks.length - 1), // latest track
152       0, // initialize with 0 stake
153       false, // add or sub does not matter
154       false, // initialize as not disabled
155       false // do not bump finished sale counter
156     );
157
158     // emit
159     emit AddTrack(name, address(stakeToken));
160   }
```

图 3 addTrack函数源码

在addTrackCheckpoint函数中，由于该track是新创建，所以仅会添加一个新的TrackCheckpoint快照记录。

```
557  function addTrackCheckpoint(
558      uint24 trackId, // track number
559      uint104 amount, // delta on staked amount
560      bool addElseSub, // true = adding; false = subtracting
561      bool disabled, // whether track is disabled; cannot undo a disable
562      bool _bumpSaleCounter // whether to increase sale counter by 1
563  ) internal {
564      // get track info
565      TrackInfo storage track = tracks[trackId];
566
567      // get track checkpoint count
568      uint32 nCheckpoints = trackCheckpointCounts[trackId];
569
570      // if this is first checkpoint
571      if (nCheckpoints == 0) {
572          // add a first checkpoint for this track
573          trackCheckpoints[trackId][0] = TrackCheckpoint({
574              blockNumber: uint80(block.number),
575              totalStaked: amount,
576              totalStakeWeight: 0,
577              disabled: disabled,
578              numFinishedSales: _bumpSaleCounter ? 1 : 0
579          });
580
581          // increase new track's checkpoint count by 1
582          trackCheckpointCounts[trackId]++;
583      } else { ···
679      }
680
681      // emit
682      emit AddTrackCheckpoint(trackId, uint80(block.number));
683  }
```

图 4 addTrackCheckpoint函数源码

- **相关函数：**addTrack、addTrackCheckpoint
- **安全建议：无**
- **审计结果：通过**

3.2.3 禁用指定track

- **业务描述：**合约的所有者 owner 可调用 disableTrack 函数将指定的 track 设置为禁用，该函数同样会调用 addTrackCheckpoint 函数更新数据，并将该 track 的最新记录中的 disabled 字段设置为 true。但需要注意，被禁用的 track 不可再次启用。
- **相关函数：**disableTrack、addTrackCheckpoint
- **安全建议：无**
- **审计结果：通过**

3.2.4 提交指定track已完成的阶段

- **业务描述：**合约的所有者 owner 可调用 bumpSaleCounter 函数更新指定 track 的 sale counter。该函数会记录完成时的区块号，并调用 addTrackCheckpoint 函数更新该 track 的 nFinishedSales。

```
163   function bumpSaleCounter(uint24 trackId) external onlyOwner {
164       // get number of finished sales of this track
165       uint24 nFinishedSales =
166           trackCheckpoints[trackId][trackCheckpointCounts[trackId] - 1]
167               .numFinishedSales;
168
169       // update map that tracks block numbers of finished sales
170       trackFinishedSaleBlocks[trackId][nFinishedSales] = uint80(block.number);
171
172       // add a new checkpoint with counter incremented by 1
173       addTrackCheckpoint(trackId, 0, false, false, true);
174
175       // `BumpSaleCounter` event emitted in function call above
176   }
```

图 5 bumpSaleCounter 函数源码

- **相关函数：**bumpSaleCounter、addTrackCheckpoint
- **安全建议：无**
- **审计结果：通过**

3.2.5 质押代币

- **业务描述：**持有指定代币的用户可调用合约的 stake 函数在指定 track 上进行质押。如下图所示，该函数要求对应的 track 未被禁用；且需要调用者先向本合约授权质押代币。同样，该函数也会调用 addUserCheckpoint 函数以新的质押总量更新对应的快照数据，并且用户总抵押量不能超过 maxTotalStake；调用 addTrackCheckpoint 函数更新该 track 的全局快照数据。

```
686   function stake(uint24 trackId, uint104 amount) external nonReentrant {
687       // stake amount must be greater than 0
688       require(amount > 0, 'amount is 0');
689
690       // get track info
691       TrackInfo storage track = tracks[trackId];
692
693       // get latest track checkpoint
694       TrackCheckpoint storage checkpoint =
695           trackCheckpoints[trackId][trackCheckpointCounts[trackId] - 1];
696
697       // cannot stake into disabled track
698       require(!checkpoint.disabled, 'track is disabled');
699
700       // transfer the specified amount of stake token from user to this contract
701       track.stakeToken.safeTransferFrom(_msgSender(), address(this), amount);
702
703       // add user checkpoint
704       addUserCheckpoint(trackId, amount, true);
705
706       // add track checkpoint
707       addTrackCheckpoint(trackId, amount, true, false, false);
708
709       // emit
710       emit Stake(trackId, _msgSender(), amount);
711   }
```

图 6 stake函数源码

- **相关函数：**stake、addUserCheckpoin、addTrackCheckpoint
- **安全建议：**无
- **审计结果：通过**

3.2.6 提取质押代币

- **业务描述：**已质押用户可调用合约的unstake函数将指定track上的质押代币提出，如下图所示，该函数首先会检查提取的数量，要求对应的提取数量大于0，且不大于该用户总的质押量；然后发送对应代币至用户地址；最后是调用addUserCheckpoint和addTrackCheckpoint函数更新对应的记录数据。

```
714  function unstake(uint24 trackId, uint104 amount) external nonReentrant {
715      // amount must be greater than 0
716      require(amount > 0, 'amount is 0');
717
718      // get track info
719      TrackInfo storage track = tracks[trackId];
720
721      // get number of user's checkpoints within this track
722      uint32 userCheckpointCount =
723          userCheckpointCounts[trackId][_msgSender()];
724
725      // get user's latest checkpoint
726      UserCheckpoint storage checkpoint =
727          userCheckpoints[trackId][_msgSender()][userCheckpointCount - 1];
728
729      // ensure amount <= user's current stake
730      require(amount <= checkpoint.staked, 'amount > staked');
731
732      // add user checkpoint
733      addUserCheckpoint(trackId, amount, false);
734
735      // add track checkpoint
736      addTrackCheckpoint(trackId, amount, false, false, false);
737
738      // transfer the specified amount of stake token from this contract to user
739      track.stakeToken.safeTransfer(_msgSender(), amount);
740
741      // emit
742      emit Unstake(trackId, _msgSender(), amount);
743  }
```

图 7 unstake函数源码

- **相关函数：**unstake、addUserCheckpoint、addTrackCheckpoint
- **安全建议：**无
- **审计结果：通过**

3.2.7 获取指定区块最近的记录数据

- **业务描述：**本合约中的getClosestUserCheckpoint函数用于获取用户在指定track、指定区块最近的记录数据;getClosestTrackCheckpoint函数用于获取指定track在指定区块最近的记录信息。这两个函数的实现逻辑基本一致，如下图所示，如果指定的区块号大于最新记录的区块号，则返回最新记录；如果指定区块小于第一条记录的区块，则返回一个空记录；否则采用二分查找法找到最近的记录并返回对应的记录(指定区块不存在记录，则会返回向前的最近一条记录)。

```
221   ) private view returns (UserCheckpoint memory cp) {
222     // get total checkpoint count for user
223     uint32 nCheckpoints = userCheckpointCounts[trackId][user];
224
225     if (
226       userCheckpoints[trackId][user][nCheckpoints - 1].blockNumber <=
227       blockNumber
228     ) {
229       // First check most recent checkpoint
230
231       // return closest checkpoint
232       return userCheckpoints[trackId][user][nCheckpoints - 1];
233     } else if (
234       userCheckpoints[trackId][user][0].blockNumber > blockNumber
235     ) {
236       // Next check earliest checkpoint
237
238       // If specified block number is earlier than user's first checkpoint,
239       // return null checkpoint
240       return
241         UserCheckpoint({
242           blockNumber: 0,
243           staked: 0,
244           stakeWeight: 0,
245           numFinishedSales: 0
246         });
247     } else {
248       // binary search on checkpoints
249       uint32 lower = 0;
250       uint32 upper = nCheckpoints - 1;
251       while (upper > lower) {
252         uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
253         UserCheckpoint memory tempCp =
254           userCheckpoints[trackId][user][center];
255         if (tempCp.blockNumber == blockNumber) {
256           return tempCp;
257         } else if (tempCp.blockNumber < blockNumber) {
258           lower = center;
259         } else {
260           upper = center - 1;
261         }
262       }
263
264       // return closest checkpoint
265       return userCheckpoints[trackId][user][lower];
266     }
267   }
```

图 8 getClosestUserCheckpoint函数源码

- **相关函数：**getClosestUserCheckpoint、getTotalStakeWeight、getClosestTrackCheckpoint

- **安全建议：无**
- **审计结果：通过**

3.2.8 获取指定区块的用户质押权重

- **业务描述：**本合约的getUserStakeWeight函数用于获取用户在指定track上、指定区块的质押权重。该函数会调用getClosestUserCheckpoint函数获取该用户在指定区块最近的记录数据，调用getClosestTrackCheckpoint函数获取全局最新的记录数据，然后通过这两条记录数据计算出用户该区块下的实际质押权重。如下图所示，当用户记录的数据与系统最新记录的数据处于同一阶段(numFinishedSales值相等)，则直接以用户记录数据中权重与新增区块所产生的累计权重来计算出新权重，并返回。

```
298     uint24 numFinishedSalesDelta =
299       closestTrackCheckpoint.numFinishedSales -
300         closestUserCheckpoint.numFinishedSales;
301
302     // get track info
303     TrackInfo memory track = tracks[trackId];
304
305     // calculate stake weight given above delta
306     uint104 stakeWeight;
307     if (numFinishedSalesDelta == 0) {
308       // calculate normally without rollover decay
309
310       uint80 elapsedBlocks =
311         blockNumber - closestUserCheckpoint.blockNumber;
312
313       stakeWeight =
314         closestUserCheckpoint.stakeWeight +
315         (uint104(elapsedBlocks) *
316           track.weightAccrualRate *
317           closestUserCheckpoint.staked) /
318         10**18;
319
320       return stakeWeight;
```

图 9 getUserStakeWeight 函数部分源码

如果用户记录数据与全局记录的数据不在同一阶段(numFinishedSales 值不相等)，则会以记录中的权重加上遍历用户在未同步阶段的累计权重，然后再加上最新阶段区块差产生的权重。

```
321     } else {
322         // calculate with rollover decay
323
324         // starting stakeweight
325         stakeWeight = closestUserCheckpoint.stakeWeight;
326         // current block for iteration
327         uint80 currBlock = closestUserCheckpoint.blockNumber;
328
329         // for each finished sale in between, get stake weight of that period
330         // and perform weighted sum
331         for (uint24 i = 0; i < numFinishedSalesDelta; i++) {
332             // get number of blocks passed at the current sale number
333             uint80 elapsedBlocks =
334                 trackFinishedSaleBlocks[trackId][
335                     closestUserCheckpoint.numFinishedSales + i
336                 ] - currBlock;
337
338             // update stake weight
339             stakeWeight =
340                 stakeWeight +
341                 (uint104(elapsedBlocks) *
342                     track.weightAccrualRate *
343                     closestUserCheckpoint.staked) /
344                 10**18;
345
346             // get amount of stake weight actively rolled over for this sale number
347             uint104 activeRolloverWeight =
348                 trackActiveRollOvers[trackId][user][
349                     closestUserCheckpoint.numFinishedSales + i
350                 ];
351
352             // factor in passive and active rollover decay
353             stakeWeight =
354                 // decay active weight
355                 (activeRolloverWeight * track.activeRolloverRate) /
356                 ROLLOVER_FACTOR_DECIMALS +
357                 // decay passive weight
358                 ((stakeWeight - activeRolloverWeight) *
359                     track.passiveRolloverRate) /
360                 ROLLOVER_FACTOR_DECIMALS;
361
362             // update currBlock for next round
363             currBlock = trackFinishedSaleBlocks[trackId][
364                 closestUserCheckpoint.numFinishedSales + i
365             ];
366         }
```

图 10 getUserStakeWeight 函数部分源码

- **相关函数：**getUserStakeWeight
- **安全建议：无**
- **审计结果：通过**

3.2.9 获取指定trackid在指定区块的系统总权重

- **业务描述：**如下图所示，本函数将获取指定track在指定区块最近全局记录数据，然后根据区块差产生的权重和记录数据的权重计算指定区块的总权重。

```
439  function getTotalStakeWeight(uint24 trackId, uint80 blockNumber)
440      external
441      view
442      returns (uint104)
443  {
444      require(blockNumber <= block.number, 'block # too high');
445
446      // get closest track checkpoint
447      TrackCheckpoint memory closestCheckpoint =
448          getClosestTrackCheckpoint(trackId, blockNumber);
449
450      // check if closest preceding checkpoint was null checkpoint
451      if (closestCheckpoint.blockNumber == 0) {
452          return 0;
453      }
454
455      // calculate blocks elapsed since checkpoint
456      uint80 additionalBlocks = (blockNumber - closestCheckpoint.blockNumber);
457
458      // get track info
459      TrackInfo storage trackInfo = tracks[trackId];
460
461      // calculate marginal accrued stake weight
462      uint104 marginalAccruedStakeWeight =
463          (uint104(additionalBlocks) *
464              trackInfo.weightAccrualRate *
465              closestCheckpoint.totalStaked) / 10**18;
466
467      // return
468      return closestCheckpoint.totalStakeWeight + marginalAccruedStakeWeight;
469  }
```

图 11 getTotalStakeWeight函数源码

- **相关函数：**getTotalStakeWeight
- **安全建议：无**
- **审计结果：通过**

3.3 IFAllocationSale合约

3.3.1 合约权限管理

- **业务描述：**本合约存在四种管理权限；owner、funder、casher 和 whitelistSetter。其中 owner 是合约所有者，可调用 emergencyTokenRetrieve 函数提取合约中的其他代币，调用

setWhitelist 函数设置 root hash 以及合约的相关设置函数对系统参数进行设置(在预售开始之前)。

```
462   function emergencyTokenRetrieve(address token) external onlyOwner {
463     // cannot be payment or sale tokens
464     require(token != address(paymentToken));
465     require(token != address(saleToken));
466
467     uint256 tokenBalance = ERC20(token).balanceOf(address(this));
468
469     // transfer all
470     ERC20(token).safeTransfer(_msgSender(), tokenBalance);
471
472     // emit
473     emit EmergencyTokenRetrieve(_msgSender(), tokenBalance);
474   }
```

图 12 emergencyTokenRetrieve 函数源码

在合约预售结束后，合约的 casher 和 owner 可调用 cash 函数提取合约接收到支付代币和多余的出售代币。**注意：如果 casher 或 owner 在所有用户都将自己购买的代币领取之前调用了 cash 函数，则未领取的用户不能领取所预购的代币。**

```
441   function cash() external onlyCasherOrOwner {
442     // must be past end block plus withdraw delay
443     require(endBlock + withdrawDelay < block.number, 'cannot withdraw yet');
444
445     // get amount of payment token received
446     uint256 paymentTokenBal = paymentToken.balanceOf(address(this));
447
448     // transfer all
449     paymentToken.safeTransfer(address(_msgSender()), paymentTokenBal);
450
451     // get amount of unsold sale token
452     uint256 saleTokenBal = saleToken.balanceOf(address(this));
453
454     // transfer all
455     saleToken.safeTransfer(address(_msgSender()), saleTokenBal);
456
457     // emit
458     emit Cash(_msgSender(), paymentTokenBal, saleTokenBal);
459   }
```

图 13 cash 函数源码

在预售开始前，合约的 funder 可调用 fund 函数向合约发送出售的代币。

```
159   function fund(uint256 amount) external onlyFunder {
160     // sale must not have started
161     require(block.number < startBlock, 'sale already started');
162
163     // transfer specified amount from funder to this contract
164     saleToken.safeTransferFrom(_msgSender(), address(this), amount);
165
166     // increase tracked sale amount
167     saleAmount += amount;
168
169     // emit
170     emit Fund(_msgSender(), amount);
171   }
```

图 14 fund 函数源码

合约的 owner 和 whitelistSetter 可调用合约的 setWhitelist 函数设置系统白名单 root hash，该操作可在预售过程中实施。

```
220   function setWhitelist(bytes32 _whitelistRootHash)
221     external
222     onlyWhitelistSetterOrOwner
223   {
224     whitelistRootHash = _whitelistRootHash;
225
226     // emit
227     emit SetWhitelist(_whitelistRootHash);
228   }
```

图 15 setWhitelist 函数源码

合约的 owner 可调用合约的 setWithdrawDelay 函数设置代币提取延时，该操作可在任意时刻实施。

```
231   function setWithdrawDelay(uint24 _withdrawDelay) external onlyOwner {
232     withdrawDelay = _withdrawDelay;
233
234     // emit
235     emit SetWithdrawDelay(_withdrawDelay);
236   }
```

图 16 setWithdrawDelay 函数源码

- **相关函数：** setMinTotalPayment、setSaleTokenAllocationOverride、setCasher、setWhitelistSetter、setWhitelist、emergencyTokenRetrieve、fund、cash
- **安全建议：无**
- **审计结果：通过**

3.3.2 普通用户购买代币

- **业务描述：**持有支付代币且在IFAllocationMaster合约上对指定track进行质押的用户，可调用本合约的purchase函数购买代币。如下图所示，函数要求当前的whitelistRootHash为空，即表示合约未开启白名单地址购买代币，然后调用_purchase进行购买代币处理。

```
357  function purchase(uint256 paymentAmount) external {
358      // there must not be a whitelist set (sales that use whitelist must be used with whitelistedPurchase)
359      require(whitelistRootHash == 0, 'use whitelistedPurchase');
360
361      _purchase(paymentAmount);
362  }
```

图 17 purchase函数源码

在_purchase函数中，会对参数和当前区块进行有效行检查；调用getMaxPayment函数获取可购买的数量；getMaxPayment函数中调用getTotalPaymentAllocation函数计算最大可以购买量；getTotalPaymentAllocation函数中调用IFAllocationMaster合约上的getUserStakeWeight函数获取调用者在指定区块(本合约指定的区块)的权重，调用getTotalStakeWeight函数获取对应track在指定区块(本合约指定的区块)的总权重，并根据权重占比计算出调用者可购买的代币数量(如果saleTokenAllocationOverride变量值不为0，则以(salePrice * saleTokenAllocationOverride) / SALE_PRICE_DECIMALS为本次的购买数量；接着对本次购买数量进行检查，并发送支付代币至本合约；最后，更新该调用者的已购买数量，避免用户购买到超过规定数量的代币。

```
318  function _purchase(uint256 paymentAmount) internal nonReentrant {
319      // sale must be active
320      require(startBlock <= block.number, 'sale has not begun');
321      require(block.number <= endBlock, 'sale over');
322
323      // sale price must not be 0, which is a giveaway sale
324      require(salePrice != 0, 'cannot purchase - giveaway sale');
325
326      // amount must be greater than minTotalPayment
327      // by default, minTotalPayment is 0 unless otherwise set
328      require(paymentAmount > minTotalPayment, 'amount below min');
329
330      // get max payment of user
331      uint256 remaining = getMaxPayment(_msgSender());
332
333      // payment must not exceed remaining
334      require(paymentAmount <= remaining, 'exceeds max payment');
335
336      // transfer specified amount from user to this contract
337      paymentToken.safeTransferFrom(
338          address(_msgSender()),
339          address(this),
340          paymentAmount
341      );
342
343      // if user is paying for the first time to this contract, increase counter
344      if (paymentReceived[_msgSender()] == 0) purchaserCount += 1;
345
346      // increase payment received amount
347      paymentReceived[_msgSender()] += paymentAmount;
348
349      // increase total payment received amount
350      totalPaymentReceived += paymentAmount;
351
352      // emit
353      emit Purchase(_msgSender(), paymentAmount);
354  }
```

图 18 _purchase函数源码

```
305  function getMaxPayment(address user) public view returns (uint256) {
306      // get the maximum total payment for a user
307      uint256 max = getTotalPaymentAllocation(user);
308      if (maxTotalPayment < max) {
309          max = maxTotalPayment;
310      }
311
312      // calculate and return remaining
313      return max - paymentReceived[user];
314  }
```

图 19 getMaxPayment 函数源码

```
254  function getTotalPaymentAllocation(address user)
255      public
256      view
257      returns (uint256)
258  {
259      // get user allocation as ratio (multiply by 10**18, aka E18, for precision)
260      uint256 userWeight = allocationMaster.getUserStakeWeight(
261          trackId,
262          user,
263          allocSnapshotBlock
264      );
265      uint256 totalWeight = allocationMaster.getTotalStakeWeight(
266          trackId,
267          allocSnapshotBlock
268      );
269
270      // total weight must be greater than 0
271      require(totalWeight > 0, 'total weight is 0');
272
273      // determine TOTAL allocation (in payment token)
274      uint256 paymentTokenAllocation;
275
276      // different calculation for whether override is set
277      if (saleTokenAllocationOverride == 0) {
278          // calculate allocation (times 10**18)
279          uint256 allocationE18 = (userWeight * 10**18) / totalWeight;
280
281          // calculate max amount of obtainable sale token
282          uint256 saleTokenAllocationE18 = (saleAmount * allocationE18);
283
284          // calculate equivalent value in payment token
285          paymentTokenAllocation =
286              (saleTokenAllocationE18 * salePrice) /
287              SALE_PRICE_DECIMALS /
288              10**18;
289      } else {
290          // override payment token allocation
291          paymentTokenAllocation =
292              (salePrice * saleTokenAllocationOverride) /
293              SALE_PRICE_DECIMALS;
294      }
295
296      return paymentTokenAllocation;
297  }
```

图 20 getTotalPaymentAllocation函数源码

- **相关函数：**purchase、_purchase
- **安全建议：无**
- **审计结果：通过**

3.3.3 白名单地址购买代币

- **业务描述：**当合约管理员调用setWhitelist函数将whitelistRootHash设置为非空时，合约将仅提供白名单地址进行代币购买。如下图所示，whitelistedPurchase函数将调用checkWhitelist函数进行白名单检查，然后调用_purchase进行代币购买。

```
365   function whitelistedPurchase(
366     uint256 paymentAmount,
367     bytes32[] calldata merkleProof
368   ) external {
369     // require that user is whitelisted by checking proof
370     require(checkWhitelist(_msgSender(), merkleProof), 'proof invalid');
371
372     _purchase(paymentAmount);
373   }
```

图 21 whitelistedPurchase函数源码

checkWhitelist函数会根据传入的参数，调用MerkleProof库内的verify函数验证地址是否是有效的白名单地址。

```
239   function checkWhitelist(address user, bytes32[] calldata merkleProof)
240     public
241     view
242     returns (bool)
243   {
244     // compute merkle leaf from input
245     bytes32 leaf = keccak256(abi.encodePacked(user));
246
247     // verify merkle proof
248     return MerkleProof.verify(merkleProof, whitelistRootHash, leaf);
249   }
```

图 22 checkWhitelist函数源码

- **相关函数：**whitelistedPurchase、_purchase、checkWhitelist
- **安全建议：无**
- **审计结果：通过**

3.3.4 提取购买的代币

- **业务描述：**提交购买代币申请的用户，在预购阶段结束后并经过一定的延迟时间（合约 owner 可以修改）后可调用合约的 withdraw 函数提取已预购的代币。该函数每个用户只能调用一次，会直接将所有预购的代币发放至用户。此外，所提取的代币售价不能为 0。

```
376   function withdraw() external nonReentrant {
377       // if there is a whitelist, an un-whitelisted user will
378       // not have any sale tokens to withdraw
379       // so we do not check whitelist here
380
381       // must be past end block plus withdraw delay
382       require(endBlock + withdrawDelay < block.number, 'cannot withdraw yet');
383       // prevent repeat withdraw
384       require(hasWithdrawn[_msgSender()] == false, 'already withdrawn');
385       // must not be a zero price sale
386       require(salePrice != 0, 'use withdrawGiveaway');
387
388       // get payment received
389       uint256 payment = paymentReceived[_msgSender()];
390
391       // calculate amount of sale token owed to buyer
392       uint256 saleTokenOwed = (payment * SALE_PRICE_DECIMALS) / salePrice;
393
394       // set withdrawn to true
395       hasWithdrawn[_msgSender()] = true;
396
397       // increment withdrawer count
398       withdrawerCount += 1;
399
400       // transfer owed sale token to buyer
401       saleToken.safeTransfer(_msgSender(), saleTokenOwed);
402
403       // emit
404       emit Withdraw(_msgSender(), saleTokenOwed);
405   }
```

图 23 withdraw 函数源码

合约实现了 withdrawGiveaway 函数用于提取售价为 0 的售卖代币。提交购买代币申请的用户，在预购阶段结束后并经过一定的延迟时间（合约 owner 可以修改）后可调用合约的 withdraw 函数提取已预购的代币。该函数每个用户只能调用一次，会直接将所有预购的代币发放至用户。

```
408  function withdrawGiveaway(bytes32[] calldata merkleProof)
409      external
410      nonReentrant
411  {
412      // must be past end block plus withdraw delay
413      require(endBlock + withdrawDelay < block.number, 'cannot withdraw yet');
414      // prevent repeat withdraw
415      require(hasWithdrawn[_msgSender()] == false, 'already withdrawn');
416      // must be a zero price sale
417      require(salePrice == 0, 'not a giveaway');
418      // if there is whitelist, require that user is whitelisted by checking proof
419      require(
420          whitelistRootHash == 0 || checkWhitelist(_msgSender(), merkleProof),
421          'proof invalid'
422      );
423
424      // each participant in the zero cost "giveaway" gets a flat amount of sale token, as set by the override
425      uint256 saleTokenOwed = saleTokenAllocationOverride;
426
427      // set withdrawn to true
428      hasWithdrawn[_msgSender()] = true;
429
430      // increment withdrawer count
431      withdrawerCount += 1;
432
433      // transfer giveaway sale token to participant
434      saleToken.safeTransfer(_msgSender(), saleTokenOwed);
435
436      // emit
437      emit Withdraw(_msgSender(), saleTokenOwed);
438  }
```

- **相关函数：**withdraw
- **安全建议：无**
- **审计结果：通过**

四、结论

Beosin(成都链安)对 launchpad 项目的智能合约的设计和代码实现进行了详细的审计。审计团队在审计过程中发现的问题均已告知项目方并就修复结果达成一致。需要注意的一点是：如果 casher 或 owner 私钥丢失并在所有用户都将自己购买的代币领取之前调用了 cash 函数，则未领取的用户不能领取所预购的代币。launchpad 项目的智能合约的总体审计结果是**通过。**

成都链安

BEOSIN

官方网址

https://lianantech.com

电子邮箱

vaas@lianantech.com

微信公众号