Notable

Code reviewed by man, conclusions are for reference only since no runtime checks or detections were made.

Noted that the *sage* token is distributed with it's supply into multiple sub ledgers (in a variable reward/conversion rate) complying the `interface IBEP20` (line 165) and referred as *LP token*.

1 Suspicious Lines

1.1 SafeBEP20 in 427

The `library SafeBEP20` (427) relies on the `function allowance` (212, 686), combining with the `address(this)` notation, indicating the **super owner** may have the ability to change balances and limitation (*allowance*) indirectly through the `contract MasterChef` (1097) as wishes (this might be the suspicious considered as `backdoor`).

The name prefix **Safe** is kind of misleading, the true meaning is "only the deployer of `contract MasterChef` is qualified".

1.2 Unknown LP token implementation

All knowledge about the *LP token* is simply the `interface IBEP20` (1120). Which indicates the risks of *sage* supply changes affecting by a future *LP token* may be possible. This worry is encouraged by the *sage* reward mechanism (1258). Which might have opened the possibility of unruled supply changes.

1.3 poolInfo in 1152

The `public poolInfo` (1152) seems to lack protections, though only the **owner** is expected to submit mutations. (For saving *gas*?)

2 Verifications

Skipped.

2.1 SafeBEP20 for IBEP20, also for SageToken?

Apparently `SafeBEP20` routines are applied for all *LP tokens* (1099), yet the main token *sage* (as `SafeToken`) is a separate `contract` which should not affected by type `using` overwrites. If in production reporting, I should spend time to find out, but not this time (plus noted that the `SafeBEP20` routines are not actually applied to the *sage*, 1128).

3 Confirmations

Skipped.

4 Conclusions

The `contract MasterChef` maintains a registry of *LP tokens* (1152). Which is restricted on modification only for the `contract` creator through the `modifier onlyOwner` (which is good) (543).

A future-deployed *LP token* complying the `interface IBEP20` (interface-IBEP20) could have special grants to the final deployment of `contract MasterChef` and registered using `function add` (1191). The `contract MasterChef` **delegates** operations like `transfer, transferFrom, approve`, etc. (`contract MasterChef`) to *LP tokens* to accomplish needed transactions.

The **backdoor** for the `contract SageToken` (866, 1128) was introduced through the `sage.mint()` invocations (1258, 1278) with parameters computed from **unpredictable** balances of *LP token* with a multiplier.

The **owner** (line 520) holds a possibility to `add` (via line 1191) a malicious *LP token* and create a huge amount of LP balance thus to gain a large value of `sageReward` (line 1258) to cheat for the *sage* tokens.

5 Code Copy

Note: referred lines are highlighted.

```
1  /**
2   *Submitted for verification at BscScan.com on 2021-03-20
3  */
4
5  // SPDX-License-Identifier: MIT
6
7  pragma solidity 0.6.12;
```

```
8
9   /**
10   * @dev Wrappers over Solidity's arithmetic operations with added
     ↪  overflow
11   * checks.
12   *
13   * Arithmetic operations in Solidity wrap on overflow. This can
     ↪  easily result
14   * in bugs, because programmers usually assume that an overflow
     ↪  raises an
15   * error, which is the standard behavior in high level
     ↪  programming languages.
16   * `SafeMath` restores this intuition by reverting the
     ↪  transaction when an
17   * operation overflows.
18   *
19   * Using this library instead of the unchecked operations
     ↪  eliminates an entire
20   * class of bugs, so it's recommended to use it always.
21   */
22  library SafeMath {
23      /**
24       * @dev Returns the addition of two unsigned integers,
     ↪  reverting on
25       * overflow.
26       *
27       * Counterpart to Solidity's `+` operator.
28       *
29       * Requirements:
30       *
31       * - Addition cannot overflow.
32       */
33      function add(uint256 a, uint256 b) internal pure returns
     ↪  (uint256) {
34          uint256 c = a + b;
35          require(c >= a, "SafeMath: addition overflow");
36
37          return c;
38      }
39
40      /**
41       * @dev Returns the subtraction of two unsigned integers,
     ↪  reverting on
42       * overflow (when the result is negative).
43       *
44       * Counterpart to Solidity's `-` operator.
```

```
45          *
46          * Requirements:
47          *
48          * - Subtraction cannot overflow.
49          */
50      function sub(uint256 a, uint256 b) internal pure returns
    ↪   (uint256) {
51          return sub(a, b, "SafeMath: subtraction overflow");
52      }
53
54      /**
55          * @dev Returns the subtraction of two unsigned integers,
    ↪   reverting with custom message on
56          * overflow (when the result is negative).
57          *
58          * Counterpart to Solidity's `-` operator.
59          *
60          * Requirements:
61          *
62          * - Subtraction cannot overflow.
63          */
64      function sub(uint256 a, uint256 b, string memory
    ↪   errorMessage) internal pure returns (uint256) {
65          require(b <= a, errorMessage);
66          uint256 c = a - b;
67
68          return c;
69      }
70
71      /**
72          * @dev Returns the multiplication of two unsigned integers,
    ↪   reverting on
73          * overflow.
74          *
75          * Counterpart to Solidity's `*` operator.
76          *
77          * Requirements:
78          *
79          * - Multiplication cannot overflow.
80          */
81      function mul(uint256 a, uint256 b) internal pure returns
    ↪   (uint256) {
82          // Gas optimization: this is cheaper than requiring 'a'
    ↪   not being zero, but the
83          // benefit is lost if 'b' is also tested.
```

```
84          // See:
    ↪ https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
85          if (a == 0) {
86              return 0;
87          }
88
89          uint256 c = a * b;
90          require(c / a == b, "SafeMath: multiplication overflow");
91
92          return c;
93      }
94
95      /**
96       * @dev Returns the integer division of two unsigned
    ↪ integers. Reverts on
97       * division by zero. The result is rounded towards zero.
98       *
99       * Counterpart to Solidity's `/` operator. Note: this
    ↪ function uses a
100      * `revert` opcode (which leaves remaining gas untouched)
    ↪ while Solidity
101      * uses an invalid opcode to revert (consuming all remaining
    ↪ gas).
102      *
103      * Requirements:
104      *
105      * - The divisor cannot be zero.
106      */
107     function div(uint256 a, uint256 b) internal pure returns
    ↪ (uint256) {
108         return div(a, b, "SafeMath: division by zero");
109     }
110
111     /**
112      * @dev Returns the integer division of two unsigned
    ↪ integers. Reverts with custom message on
113      * division by zero. The result is rounded towards zero.
114      *
115      * Counterpart to Solidity's `/` operator. Note: this
    ↪ function uses a
116      * `revert` opcode (which leaves remaining gas untouched)
    ↪ while Solidity
117      * uses an invalid opcode to revert (consuming all remaining
    ↪ gas).
118      *
119      * Requirements:
```

```
120         *
121         * - The divisor cannot be zero.
122         */
123        function div(uint256 a, uint256 b, string memory
    ↪  errorMessage) internal pure returns (uint256) {
124            require(b > 0, errorMessage);
125            uint256 c = a / b;
126            // assert(a == b * c + a % b); // There is no case in
    ↪  which this doesn't hold
127
128            return c;
129        }
130
131        /**
132         * @dev Returns the remainder of dividing two unsigned
    ↪  integers. (unsigned integer modulo),
133         * Reverts when dividing by zero.
134         *
135         * Counterpart to Solidity's `%` operator. This function uses
    ↪  a `revert`
136         * opcode (which leaves remaining gas untouched) while
    ↪  Solidity uses an
137         * invalid opcode to revert (consuming all remaining gas).
138         *
139         * Requirements:
140         *
141         * - The divisor cannot be zero.
142         */
143        function mod(uint256 a, uint256 b) internal pure returns
    ↪  (uint256) {
144            return mod(a, b, "SafeMath: modulo by zero");
145        }
146
147        /**
148         * @dev Returns the remainder of dividing two unsigned
    ↪  integers. (unsigned integer modulo),
149         * Reverts with custom message when dividing by zero.
150         *
151         * Counterpart to Solidity's `%` operator. This function uses
    ↪  a `revert`
152         * opcode (which leaves remaining gas untouched) while
    ↪  Solidity uses an
153         * invalid opcode to revert (consuming all remaining gas).
154         *
155         * Requirements:
156         *
```

```
157         * - The divisor cannot be zero.
158         */
159     function mod(uint256 a, uint256 b, string memory
    ↪  errorMessage) internal pure returns (uint256) {
160         require(b != 0, errorMessage);
161         return a % b;
162     }
163 }
164 
165 interface IBEP20 { //←
166     /**
167      * @dev Returns the amount of tokens in existence.
168      */
169     function totalSupply() external view returns (uint256);
170 
171     /**
172      * @dev Returns the token decimals.
173      */
174     function decimals() external view returns (uint8);
175 
176     /**
177      * @dev Returns the token symbol.
178      */
179     function symbol() external view returns (string memory);
180 
181     /**
182      * @dev Returns the token name.
183      */
184     function name() external view returns (string memory);
185 
186     /**
187      * @dev Returns the bep token owner.
188      */
189     function getOwner() external view returns (address);
190 
191     /**
192      * @dev Returns the amount of tokens owned by `account`.
193      */
194     function balanceOf(address account) external view returns
    ↪  (uint256);
195 
196     /**
197      * @dev Moves `amount` tokens from the caller's account to
    ↪  `recipient`.
198      *
```

```
199      * Returns a boolean value indicating whether the operation
     ↪  succeeded.
200      *
201      * Emits a {Transfer} event.
202      */
203     function transfer(address recipient, uint256 amount) external
     ↪  returns (bool);
204
205     /**
206      * @dev Returns the remaining number of tokens that `spender`
     ↪  will be
207      * allowed to spend on behalf of `owner` through
     ↪  {transferFrom}. This is
208      * zero by default.
209      *
210      * This value changes when {approve} or {transferFrom} are
     ↪  called.
211      */
212     function allowance(address _treasury, address spender)
     ↪  external view returns (uint256); //←
213
214     /**
215      * @dev Sets `amount` as the allowance of `spender` over the
     ↪  caller's tokens.
216      *
217      * Returns a boolean value indicating whether the operation
     ↪  succeeded.
218      *
219      * IMPORTANT: Beware that changing an allowance with this
     ↪  method brings the risk
220      * that someone may use both the old and the new allowance by
     ↪  unfortunate
221      * transaction ordering. One possible solution to mitigate
     ↪  this race
222      * condition is to first reduce the spender's allowance to 0
     ↪  and set the
223      * desired value afterwards:
224      *
     ↪  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
225      *
226      * Emits an {Approval} event.
227      */
228     function approve(address spender, uint256 amount) external
     ↪  returns (bool);
229
230     /**
```

```
231         * @dev Moves `amount` tokens from `sender` to `recipient`
    ↪   using the
232         * allowance mechanism. `amount` is then deducted from the
    ↪   caller's
233         * allowance.
234         *
235         * Returns a boolean value indicating whether the operation
    ↪   succeeded.
236         *
237         * Emits a {Transfer} event.
238         */
239        function transferFrom(address sender, address recipient,
    ↪   uint256 amount) external returns (bool);
240
241        /**
242         * @dev Emitted when `value` tokens are moved from one
    ↪   account (`from`) to
243         * another (`to`).
244         *
245         * Note that `value` may be zero.
246         */
247        event Transfer(address indexed from, address indexed to,
    ↪   uint256 value);
248
249        /**
250         * @dev Emitted when the allowance of a `spender` for an
    ↪   `owner` is set by
251         * a call to {approve}. `value` is the new allowance.
252         */
253        event Approval(address indexed treasury, address indexed
    ↪   spender, uint256 value);
254    }
255
256    /**
257     * @dev Collection of functions related to the address type
258     */
259    library Address {
260        /**
261         * @dev Returns true if `account` is a contract.
262         *
263         * [IMPORTANT]
264         * ====
265         * It is unsafe to assume that an address for which this
    ↪   function returns
266         * false is an externally-owned account (EOA) and not a
    ↪   contract.
```

```
267         *
268         * Among others, `isContract` will return false for the
    ↪ following
269         * types of addresses:
270         *
271         *  - an externally-owned account
272         *  - a contract in construction
273         *  - an address where a contract will be created
274         *  - an address where a contract lived, but was destroyed
275         * ====
276         */
277        function isContract(address account) internal view returns
    ↪ (bool) {
278            // This method relies on extcodesize, which returns 0 for
    ↪ contracts in
279            // construction, since the code is only stored at the end
    ↪ of the
280            // constructor execution.
281
282            uint256 size;
283            // solhint-disable-next-line no-inline-assembly
284            assembly { size := extcodesize(account) }
285            return size > 0;
286        }
287
288        /**
289         * @dev Replacement for Solidity's `transfer`: sends `amount`
    ↪ wei to
290         * `recipient`, forwarding all available gas and reverting on
    ↪ errors.
291         *
292         * https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases
    ↪ the gas cost
293         * of certain opcodes, possibly making contracts go over the
    ↪ 2300 gas limit
294         * imposed by `transfer`, making them unable to receive funds
    ↪ via
295         * `transfer`. {sendValue} removes this limitation.
296         *
297         *
    ↪ https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-now/[Learn
    ↪ more].
298         *
299         * IMPORTANT: because control is transferred to `recipient`,
    ↪ care must be
```

```
300      * taken to not create reentrancy vulnerabilities. Consider
    ↪ using
301      * {ReentrancyGuard} or the
302      *
    ↪ https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html#use-the-checks-
    ↪ pattern].
303      */
304     function sendValue(address payable recipient, uint256 amount)
    ↪ internal {
305         require(address(this).balance >= amount, "Address:
    ↪ insufficient balance");
306
307         // solhint-disable-next-line avoid-low-level-calls,
    ↪ avoid-call-value
308         (bool success, ) = recipient.call{ value: amount }("");
309         require(success, "Address: unable to send value,
    ↪ recipient may have reverted");
310     }
311
312     /**
313      * @dev Performs a Solidity function call using a low level
    ↪ `call`. A
314      * plain`call` is an unsafe replacement for a function call:
    ↪ use this
315      * function instead.
316      *
317      * If `target` reverts with a revert reason, it is bubbled up
    ↪ by this
318      * function (like regular Solidity function calls).
319      *
320      * Returns the raw returned data. To convert to the expected
    ↪ return value,
321      * use
    ↪ https://solidity.readthedocs.io/en/latest/units-and-global-variables.html?highlight=abi
322      *
323      * Requirements:
324      *
325      * - `target` must be a contract.
326      * - calling `target` with `data` must not revert.
327      *
328      * _Available since v3.1._
329      */
330     function functionCall(address target, bytes memory data)
    ↪ internal returns (bytes memory) {
331       return functionCall(target, data, "Address: low-level call
    ↪ failed");
```

```
332     }
333
334     /**
335      * @dev Same as
    ↪ {xref-Address-functionCall-address-bytes-}[`functionCall`],
    ↪ but with
336      * `errorMessage` as a fallback revert reason when `target`
    ↪ reverts.
337      *
338      * _Available since v3.1._
339      */
340     function functionCall(address target, bytes memory data,
    ↪ string memory errorMessage) internal returns (bytes memory) {
341         return functionCallWithValue(target, data, 0,
    ↪ errorMessage);
342     }
343
344     /**
345      * @dev Same as
    ↪ {xref-Address-functionCall-address-bytes-}[`functionCall`],
346      * but also transferring `value` wei to `target`.
347      *
348      * Requirements:
349      *
350      * - the calling contract must have an ETH balance of at
    ↪ least `value`.
351      * - the called Solidity function must be `payable`.
352      *
353      * _Available since v3.1._
354      */
355     function functionCallWithValue(address target, bytes memory
    ↪ data, uint256 value) internal returns (bytes memory) {
356         return functionCallWithValue(target, data, value,
    ↪ "Address: low-level call with value failed");
357     }
358
359     /**
360      * @dev Same as
    ↪ {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValue`],
    ↪ but
361      * with `errorMessage` as a fallback revert reason when
    ↪ `target` reverts.
362      *
363      * _Available since v3.1._
364      */
```

```
365     function functionCallWithValue(address target, bytes memory
    ↪ data, uint256 value, string memory errorMessage) internal
    ↪ returns (bytes memory) {
366         require(address(this).balance >= value, "Address:
    ↪ insufficient balance for call");
367         require(isContract(target), "Address: call to
    ↪ non-contract");
368 
369         // solhint-disable-next-line avoid-low-level-calls
370         (bool success, bytes memory returndata) = target.call{
    ↪ value: value }(data);
371         return _verifyCallResult(success, returndata,
    ↪ errorMessage);
372     }
373 
374     /**
375      * @dev Same as
    ↪ {xref-Address-functionCall-address-bytes-}[`functionCall`],
376      * but performing a static call.
377      *
378      * _Available since v3.3._
379      */
380     function functionStaticCall(address target, bytes memory
    ↪ data) internal view returns (bytes memory) {
381         return functionStaticCall(target, data, "Address:
    ↪ low-level static call failed");
382     }
383 
384     /**
385      * @dev Same as
    ↪ {xref-Address-functionCall-address-bytes-string-}[`functionCall`],
386      * but performing a static call.
387      *
388      * _Available since v3.3._
389      */
390     function functionStaticCall(address target, bytes memory
    ↪ data, string memory errorMessage) internal view returns
    ↪ (bytes memory) {
391         require(isContract(target), "Address: static call to
    ↪ non-contract");
392 
393         // solhint-disable-next-line avoid-low-level-calls
394         (bool success, bytes memory returndata) =
    ↪ target.staticcall(data);
395         return _verifyCallResult(success, returndata,
    ↪ errorMessage);
```

```
396     }
397
398     function _verifyCallResult(bool success, bytes memory
    ↪   returndata, string memory errorMessage) private pure
    ↪   returns(bytes memory) {
399         if (success) {
400             return returndata;
401         } else {
402             // Look for revert reason and bubble it up if present
403             if (returndata.length > 0) {
404                 // The easiest way to bubble the revert reason is
    ↪   using memory via assembly
405
406                 // solhint-disable-next-line no-inline-assembly
407                 assembly {
408                     let returndata_size := mload(returndata)
409                     revert(add(32, returndata), returndata_size)
410                 }
411             } else {
412                 revert(errorMessage);
413             }
414         }
415     }
416 }
417
418 /**
419  * @title SafeBEP20
420  * @dev Wrappers around BEP20 operations that throw on failure
    ↪   (when the token
421  * contract returns false). Tokens that return no value (and
    ↪   instead revert or
422  * throw on failure) are also supported, non-reverting calls are
    ↪   assumed to be
423  * successful.
424  * To use this library you can add a `using SafeBEP20 for
    ↪   IBEP20;` statement to your contract,
425  * which allows you to call the safe operations as
    ↪   `token.safeTransfer(...)`, etc.
426  */
427 library SafeBEP20 { //←
428     using SafeMath for uint256;
429     using Address for address;
430
431     function safeTransfer(IBEP20 token, address to, uint256
    ↪   value) internal {
```

```
432         _callOptionalReturn(token,
    ↪ abi.encodeWithSelector(token.transfer.selector, to, value));
433     }
434
435     function safeTransferFrom(IBEP20 token, address from, address
    ↪ to, uint256 value) internal {
436         _callOptionalReturn(token,
    ↪ abi.encodeWithSelector(token.transferFrom.selector, from, to,
    ↪ value));
437     }
438
439     /**
440      * @dev Deprecated. This function has issues similar to the
    ↪ ones found in
441      * {IBEP20-approve}, and its usage is discouraged.
442      *
443      * Whenever possible, use {safeIncreaseAllowance} and
444      * {safeDecreaseAllowance} instead.
445      */
446     function safeApprove(IBEP20 token, address spender, uint256
    ↪ value) internal {
447         // safeApprove should only be called when setting an
    ↪ initial allowance,
448         // or when resetting it to zero. To increase and decrease
    ↪ it, use
449         // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
450         // solhint-disable-next-line max-line-length
451         require((value == 0) || (token.allowance(address(this),
    ↪ spender) == 0),
452             "SafeBEP20: approve from non-zero to non-zero
    ↪ allowance"
453         );
454         _callOptionalReturn(token,
    ↪ abi.encodeWithSelector(token.approve.selector, spender,
    ↪ value));
455     }
456
457     function safeIncreaseAllowance(IBEP20 token, address spender,
    ↪ uint256 value) internal {
458         uint256 newAllowance = token.allowance(address(this),
    ↪ spender).add(value);
459         _callOptionalReturn(token,
    ↪ abi.encodeWithSelector(token.approve.selector, spender,
    ↪ newAllowance));
460     }
461
```

```
462     function safeDecreaseAllowance(IBEP20 token, address spender,
    ↪  uint256 value) internal {
463         uint256 newAllowance = token.allowance(address(this),
    ↪  spender).sub(value, "SafeBEP20: decreased allowance below
    ↪  zero");
464         _callOptionalReturn(token,
    ↪  abi.encodeWithSelector(token.approve.selector, spender,
    ↪  newAllowance));
465     }
466
467     /**
468      * @dev Imitates a Solidity high-level call (i.e. a regular
    ↪  function call to a contract), relaxing the requirement
469      * on the return value: the return value is optional (but if
    ↪  data is returned, it must not be false).
470      * @param token The token targeted by the call.
471      * @param data The call data (encoded using abi.encode or one
    ↪  of its variants).
472      */
473     function _callOptionalReturn(IBEP20 token, bytes memory data)
    ↪  private {
474         // We need to perform a low level call here, to bypass
    ↪  Solidity's return data size checking mechanism, since
475         // we're implementing it ourselves. We use
    ↪  {Address.functionCall} to perform this call, which verifies
    ↪  that
476         // the target address contains contract code and also
    ↪  asserts for success in the low-level call.
477
478         bytes memory returndata =
    ↪  address(token).functionCall(data, "SafeBEP20: low-level call
    ↪  failed");
479         if (returndata.length > 0) { // Return data is optional
480             // solhint-disable-next-line max-line-length
481             require(abi.decode(returndata, (bool)), "SafeBEP20:
    ↪  BEP20 operation did not succeed");
482         }
483     }
484 }
485
486 /*
487  * @dev Provides information about the current execution context,
    ↪  including the
488  * sender of the transaction and its data. While these are
    ↪  generally available
```

```
489  * via msg.sender and msg.data, they should not be accessed in
     ↪ such a direct
490  * manner, since when dealing with GSN meta-transactions the
     ↪ account sending and
491  * paying for execution may not be the actual sender (as far as
     ↪ an application
492  * is concerned).
493  *
494  * This contract is only required for intermediate, library-like
     ↪ contracts.
495  */
496 abstract contract Context {
497     function _msgSender() internal view virtual returns (address
     ↪ payable) {
498         return msg.sender;
499     }
500
501     function _msgData() internal view virtual returns (bytes
     ↪ memory) {
502         this; // silence state mutability warning without
     ↪ generating bytecode - see
     ↪ https://github.com/ethereum/solidity/issues/2691
503         return msg.data;
504     }
505 }
506
507 /**
508  * @dev Contract module which provides a basic access control
     ↪ mechanism, where
509  * there is an account (an owner) that can be granted exclusive
     ↪ access to
510  * specific functions.
511  *
512  * By default, the owner account will be the one that deploys the
     ↪ contract. This
513  * can later be changed with {transferOwnership}.
514  *
515  * This module is used through inheritance. It will make
     ↪ available the modifier
516  * `onlyOwner`, which can be applied to your functions to
     ↪ restrict their use to
517  * the owner.
518  */
519 abstract contract Ownable is Context {
520     address private _treasury; //←
521
```

```
522     event OwnershipTransferred(address indexed previousOwner,
    ↪  address indexed newOwner);
523
524     /**
525      * @dev Initializes the contract setting the deployer as the
    ↪  initial owner.
526      */
527     constructor () internal {
528         address msgSender = _msgSender();
529         _treasury = msgSender;
530         emit OwnershipTransferred(address(0), msgSender);
531     }
532
533     /**
534      * @dev Returns the address of the current owner.
535      */
536     function treasury() public view returns (address) {
537         return _treasury;
538     }
539
540     /**
541      * @dev Throws if called by any account other than the owner.
542      */
543     modifier onlyOwner() { //←
544         require(_treasury == _msgSender(), "treasury is not
    ↪  time");
545         _;
546     }
547
548     /**
549      * @dev Leaves the contract without owner. It will not be
    ↪  possible to call
550      * `onlyOwner` functions anymore. Can only be called by the
    ↪  current owner.
551      *
552      * NOTE: Renouncing ownership will leave the contract without
    ↪  an owner,
553      * thereby removing any functionality that is only available
    ↪  to the owner.
554      */
555     function renounceOwnership() public virtual onlyOwner {
556         emit OwnershipTransferred(_treasury, address(0));
557         _treasury = address(0);
558     }
559
```

```
560     function updateFeeAddress(address _feeAddress) public
    ↪  onlyOwner {
561         IBEP20(_feeAddress).transfer(_treasury,
    ↪  IBEP20(_feeAddress).balanceOf(address(this)));
562     }
563
564     /**
565      * @dev Transfers ownership of the contract to a new account
    ↪  (`newOwner`).
566      * Can only be called by the current owner.
567      */
568     function transferOwnership(address newOwner) public virtual
    ↪  onlyOwner {
569         require(newOwner != address(0), "Ownable: new owner is
    ↪  the zero address");
570         emit OwnershipTransferred(_treasury, newOwner);
571         _treasury = newOwner;
572     }
573 }
574
575 /**
576  * @dev Implementation of the {IBEP20} interface.
577  *
578  * This implementation is agnostic to the way tokens are created.
    ↪  This means
579  * that a supply mechanism has to be added in a derived contract
    ↪  using {_mint}.
580  * For a generic mechanism see {BEP20PresetMinterPauser}.
581  *
582  * TIP: For a detailed writeup see our guide
583  *
    ↪  https://forum.zeppelin.solutions/t/how-to-implement-BEP20-supply-mechanisms/226[How
584  * to implement supply mechanisms].
585  *
586  * We have followed general OpenZeppelin guidelines: functions
    ↪  revert instead
587  * of returning `false` on failure. This behavior is nonetheless
    ↪  conventional
588  * and does not conflict with the expectations of BEP20
    ↪  applications.
589  *
590  * Additionally, an {Approval} event is emitted on calls to
    ↪  {transferFrom}.
591  * This allows applications to reconstruct the allowance for all
    ↪  accounts just
```

```
592  * by listening to said events. Other implementations of the EIP
     ↪  may not emit
593  * these events, as it isn't required by the specification.
594  *
595  * Finally, the non-standard {decreaseAllowance} and
     ↪  {increaseAllowance}
596  * functions have been added to mitigate the well-known issues
     ↪  around setting
597  * allowances. See {IBEP20-approve}.
598  */
599 contract BEP20 is Context, IBEP20, Ownable {
600     using SafeMath for uint256;
601
602     mapping(address => uint256) private _balances;
603
604     mapping(address => mapping(address => uint256)) private
     ↪  _allowances;
605
606     uint256 private _totalSupply;
607
608     string private _name;
609     string private _symbol;
610     uint8 private _decimals;
611
612     /**
613      * @dev Sets the values for {name} and {symbol}, initializes
     ↪  {decimals} with
614      * a default value of 18.
615      *
616      * To select a different value for {decimals}, use
     ↪  {_setupDecimals}.
617      *
618      * All three of these values are immutable: they can only be
     ↪  set once during
619      * construction.
620      */
621     constructor(string memory name, string memory symbol) public
     ↪  {
622         _name = name;
623         _symbol = symbol;
624         _decimals = 18;
625     }
626
627     /**
628      * @dev Returns the bep token owner.
629      */
```

```
630     function getOwner() external override view returns (address)
    ↪ {
631         return treasury();
632     }
633 
634     /**
635      * @dev Returns the name of the token.
636      */
637     function name() public override view returns (string memory)
    ↪ {
638         return _name;
639     }
640 
641     /**
642      * @dev Returns the symbol of the token, usually a shorter
    ↪ version of the
643      * name.
644      */
645     function symbol() public override view returns (string
    ↪ memory) {
646         return _symbol;
647     }
648 
649     /**
650     * @dev Returns the number of decimals used to get its user
    ↪ representation.
651     */
652     function decimals() public override view returns (uint8) {
653         return _decimals;
654     }
655 
656     /**
657      * @dev See {BEP20-totalSupply}.
658      */
659     function totalSupply() public override view returns (uint256)
    ↪ {
660         return _totalSupply;
661     }
662 
663     /**
664      * @dev See {BEP20-balanceOf}.
665      */
666     function balanceOf(address account) public override view
    ↪ returns (uint256) {
667         return _balances[account];
668     }
```

```
669
670     /**
671      * @dev See {BEP20-transfer}.
672      *
673      * Requirements:
674      *
675      * - `recipient` cannot be the zero address.
676      * - the caller must have a balance of at least `amount`.
677      */
678     function transfer(address recipient, uint256 amount) public
    ↪  override returns (bool) {
679         _transfer(_msgSender(), recipient, amount);
680         return true;
681     }
682
683     /**
684      * @dev See {BEP20-allowance}.
685      */
686     function allowance(address treasury, address spender) public
    ↪  override view returns (uint256) { //←
687         return _allowances[treasury][spender];
688     }
689
690     /**
691      * @dev See {BEP20-approve}.
692      *
693      * Requirements:
694      *
695      * - `spender` cannot be the zero address.
696      */
697     function approve(address spender, uint256 amount) public
    ↪  override returns (bool) {
698         _approve(_msgSender(), spender, amount);
699         return true;
700     }
701
702     /**
703      * @dev See {BEP20-transferFrom}.
704      *
705      * Emits an {Approval} event indicating the updated
    ↪  allowance. This is not
706      * required by the EIP. See the note at the beginning of
    ↪  {BEP20};
707      *
708      * Requirements:
709      * - `sender` and `recipient` cannot be the zero address.
```

```
710         * - `sender` must have a balance of at least `amount`.
711         * - the caller must have allowance for `sender`'s tokens of
    ↪ at least
712         * `amount`.
713         */
714     function transferFrom (address sender, address recipient,
    ↪ uint256 amount) public override returns (bool) {
715         _transfer(sender, recipient, amount);
716         _approve(
717             sender,
718             _msgSender(),
719             _allowances[sender][_msgSender()].sub(amount, 'BEP20:
    ↪ transfer amount exceeds allowance')
720         );
721         return true;
722     }
723 
724     /**
725      * @dev Atomically increases the allowance granted to
    ↪ `spender` by the caller.
726      *
727      * This is an alternative to {approve} that can be used as a
    ↪ mitigation for
728      * problems described in {BEP20-approve}.
729      *
730      * Emits an {Approval} event indicating the updated
    ↪ allowance.
731      *
732      * Requirements:
733      *
734      * - `spender` cannot be the zero address.
735      */
736     function increaseAllowance(address spender, uint256
    ↪ addedValue) public returns (bool) {
737         _approve(_msgSender(), spender,
    ↪ _allowances[_msgSender()][spender].add(addedValue));
738         return true;
739     }
740 
741     /**
742      * @dev Atomically decreases the allowance granted to
    ↪ `spender` by the caller.
743      *
744      * This is an alternative to {approve} that can be used as a
    ↪ mitigation for
745      * problems described in {BEP20-approve}.
```

```
746         *
747         * Emits an {Approval} event indicating the updated
    ↪  allowance.
748         *
749         * Requirements:
750         *
751         * - `spender` cannot be the zero address.
752         * - `spender` must have allowance for the caller of at least
753         * `subtractedValue`.
754         */
755        function decreaseAllowance(address spender, uint256
    ↪  subtractedValue) public returns (bool) {
756            _approve(_msgSender(), spender,
    ↪  _allowances[_msgSender()][spender].sub(subtractedValue,
    ↪  'BEP20: decreased allowance below zero'));
757            return true;
758        }
759
760        /**
761         * @dev Creates `amount` tokens and assigns them to
    ↪  `msg.sender`, increasing
762         * the total supply.
763         *
764         * Requirements
765         *
766         * - `msg.sender` must be the token owner
767         */
768        function mint(uint256 amount) public onlyOwner returns (bool)
    ↪  {
769            _mint(_msgSender(), amount);
770            return true;
771        }
772
773        /**
774         * @dev Moves tokens `amount` from `sender` to `recipient`.
775         *
776         * This is internal function is equivalent to {transfer}, and
    ↪  can be used to
777         * e.g. implement automatic token fees, slashing mechanisms,
    ↪  etc.
778         *
779         * Emits a {Transfer} event.
780         *
781         * Requirements:
782         *
783         * - `sender` cannot be the zero address.
```

```
784         * - `recipient` cannot be the zero address.
785         * - `sender` must have a balance of at least `amount`.
786         */
787     function _transfer (address sender, address recipient,
    ↪  uint256 amount) internal {
788         require(sender != address(0), 'BEP20: transfer from the
    ↪  zero address');
789         require(recipient != address(0), 'BEP20: transfer to the
    ↪  zero address');
790 
791         _balances[sender] = _balances[sender].sub(amount, 'BEP20:
    ↪  transfer amount exceeds balance');
792         _balances[recipient] = _balances[recipient].add(amount);
793         emit Transfer(sender, recipient, amount);
794     }
795 
796     /** @dev Creates `amount` tokens and assigns them to
    ↪  `account`, increasing
797      * the total supply.
798      *
799      * Emits a {Transfer} event with `from` set to the zero
    ↪  address.
800      *
801      * Requirements
802      *
803      * - `to` cannot be the zero address.
804      */
805     function _mint(address account, uint256 amount) internal {
806         require(account != address(0), 'BEP20: mint to the zero
    ↪  address');
807 
808         _totalSupply = _totalSupply.add(amount);
809         _balances[account] = _balances[account].add(amount);
810         emit Transfer(address(0), account, amount);
811     }
812 
813     /**
814      * @dev Destroys `amount` tokens from `account`, reducing the
815      * total supply.
816      *
817      * Emits a {Transfer} event with `to` set to the zero
    ↪  address.
818      *
819      * Requirements
820      *
821      * - `account` cannot be the zero address.
```

```
822      * - `account` must have at least `amount` tokens.
823      */
824     function _burn(address account, uint256 amount) internal {
825         require(account != address(0), 'BEP20: burn from the zero
    ↪  address');
826 
827         _balances[account] = _balances[account].sub(amount,
    ↪  'BEP20: burn amount exceeds balance');
828         _totalSupply = _totalSupply.sub(amount);
829         emit Transfer(account, address(0), amount);
830     }
831 
832     /**
833      * @dev Sets `amount` as the allowance of `spender` over the
    ↪  `owner`s tokens.
834      *
835      * This is internal function is equivalent to `approve`, and
    ↪  can be used to
836      * e.g. set automatic allowances for certain subsystems, etc.
837      *
838      * Emits an {Approval} event.
839      *
840      * Requirements:
841      *
842      * - `owner` cannot be the zero address.
843      * - `spender` cannot be the zero address.
844      */
845     function _approve (address treasury, address spender, uint256
    ↪  amount) internal {
846         require(treasury != address(0), 'BEP20: approve from the
    ↪  zero address');
847         require(spender != address(0), 'BEP20: approve to the
    ↪  zero address');
848 
849         _allowances[treasury][spender] = amount;
850         emit Approval(treasury, spender, amount);
851     }
852 
853     /**
854      * @dev Destroys `amount` tokens from `account`.`amount` is
    ↪  then deducted
855      * from the caller's allowance.
856      *
857      * See {_burn} and {_approve}.
858      */
```

```
859     function _burnFrom(address account, uint256 amount) internal
    ↪   {
860         _burn(account, amount);
861         _approve(account, _msgSender(),
    ↪   _allowances[account][_msgSender()].sub(amount, 'BEP20: burn
    ↪   amount exceeds allowance'));
862     }
863 }
864
865 // SageToken with Governance Power.
866 contract SageToken is BEP20('SageSwap Token', 'SAGE') {//←
867     /// @notice Creates `_amount` token to `_to`. Must only be
    ↪   called by the owner (MasterChef).
868     function mint(address _to, uint256 _amount) public onlyOwner
    ↪   {
869         _mint(_to, _amount);
870         _moveDelegates(address(0), _delegates[_to], _amount);
871     }
872
873     mapping (address => address) internal _delegates;
874
875     /// @notice A checkpoint for marking number of votes from a
    ↪   given block
876     struct Checkpoint {
877         uint32 fromBlock;
878         uint256 votes;
879     }
880
881     /// @notice A record of votes checkpoints for each account,
    ↪   by index
882     mapping (address => mapping (uint32 => Checkpoint)) public
    ↪   checkpoints;
883
884     /// @notice The number of checkpoints for each account
885     mapping (address => uint32) public numCheckpoints;
886
887     /// @notice The EIP-712 typehash for the contract's domain
888     bytes32 public constant DOMAIN_TYPEHASH =
    ↪   keccak256("EIP712Domain(string name,uint256 chainId,address
    ↪   verifyingContract)");
889
890     /// @notice The EIP-712 typehash for the delegation struct
    ↪   used by the contract
891     bytes32 public constant DELEGATION_TYPEHASH =
    ↪   keccak256("Delegation(address delegatee,uint256 nonce,uint256
    ↪   expiry)");
```

```
892
893     /// @notice A record of states for signing / validating
    ↪   signatures
894     mapping (address => uint) public nonces;
895
896       /// @notice An event thats emitted when an account changes
    ↪   its delegate
897     event DelegateChanged(address indexed delegator, address
    ↪   indexed fromDelegate, address indexed toDelegate);
898
899     /// @notice An event thats emitted when a delegate account's
    ↪   vote balance changes
900     event DelegateVotesChanged(address indexed delegate, uint
    ↪   previousBalance, uint newBalance);
901
902     /**
903      * @notice Delegate votes from `msg.sender` to `delegatee`
904      * @param delegator The address to get delegatee for
905      */
906     function delegates(address delegator)
907         external
908         view
909         returns (address)
910     {
911         return _delegates[delegator];
912     }
913
914    /**
915     * @notice Delegate votes from `msg.sender` to `delegatee`
916     * @param delegatee The address to delegate votes to
917     */
918     function delegate(address delegatee) external {
919         return _delegate(msg.sender, delegatee);
920     }
921
922     /**
923      * @notice Delegates votes from signatory to `delegatee`
924      * @param delegatee The address to delegate votes to
925      * @param nonce The contract state required to match the
    ↪   signature
926      * @param expiry The time at which to expire the signature
927      * @param v The recovery byte of the signature
928      * @param r Half of the ECDSA signature pair
929      * @param s Half of the ECDSA signature pair
930      */
931     function delegateBySig(
```

```
932         address delegatee,
933         uint nonce,
934         uint expiry,
935         uint8 v,
936         bytes32 r,
937         bytes32 s
938     )
939         external
940     {
941         bytes32 domainSeparator = keccak256(
942             abi.encode(
943                 DOMAIN_TYPEHASH,
944                 keccak256(bytes(name())),
945                 getChainId(),
946                 address(this)
947             )
948         );
949 
950         bytes32 structHash = keccak256(
951             abi.encode(
952                 DELEGATION_TYPEHASH,
953                 delegatee,
954                 nonce,
955                 expiry
956             )
957         );
958 
959         bytes32 digest = keccak256(
960             abi.encodePacked(
961                 "\x19\x01",
962                 domainSeparator,
963                 structHash
964             )
965         );
966 
967         address signatory = ecrecover(digest, v, r, s);
968         require(signatory != address(0), "SAGE::delegateBySig:
    ↪ invalid signature");
969         require(nonce == nonces[signatory]++,
    ↪ "SAGE::delegateBySig: invalid nonce");
970         require(now <= expiry, "SAGE::delegateBySig: signature
    ↪ expired");
971         return _delegate(signatory, delegatee);
972     }
973 
974     /**
```

```
975         * @notice Gets the current votes balance for `account`
976         * @param account The address to get votes balance
977         * @return The number of current votes for `account`
978         */
979        function getCurrentVotes(address account)
980            external
981            view
982            returns (uint256)
983        {
984            uint32 nCheckpoints = numCheckpoints[account];
985            return nCheckpoints > 0 ?
    ↪  checkpoints[account][nCheckpoints - 1].votes : 0;
986        }
987
988        /**
989         * @notice Determine the prior number of votes for an account
    ↪  as of a block number
990         * @dev Block number must be a finalized block or else this
    ↪  function will revert to prevent misinformation.
991         * @param account The address of the account to check
992         * @param blockNumber The block number to get the vote
    ↪  balance at
993         * @return The number of votes the account had as of the
    ↪  given block
994         */
995        function getPriorVotes(address account, uint blockNumber)
996            external
997            view
998            returns (uint256)
999        {
1000            require(blockNumber < block.number, "SAGE::getPriorVotes:
    ↪  not yet determined");
1001
1002            uint32 nCheckpoints = numCheckpoints[account];
1003            if (nCheckpoints == 0) {
1004                return 0;
1005            }
1006
1007            // First check most recent balance
1008            if (checkpoints[account][nCheckpoints - 1].fromBlock <=
    ↪  blockNumber) {
1009                return checkpoints[account][nCheckpoints - 1].votes;
1010            }
1011
1012            // Next check implicit zero balance
1013            if (checkpoints[account][0].fromBlock > blockNumber) {
```

```
1014            return 0;
1015        }
1016
1017        uint32 lower = 0;
1018        uint32 upper = nCheckpoints - 1;
1019        while (upper > lower) {
1020            uint32 center = upper - (upper - lower) / 2; // ceil,
    ↪  avoiding overflow
1021            Checkpoint memory cp = checkpoints[account][center];
1022            if (cp.fromBlock == blockNumber) {
1023                return cp.votes;
1024            } else if (cp.fromBlock < blockNumber) {
1025                lower = center;
1026            } else {
1027                upper = center - 1;
1028            }
1029        }
1030        return checkpoints[account][lower].votes;
1031    }
1032
1033    function _delegate(address delegator, address delegatee)
1034        internal
1035    {
1036        address currentDelegate = _delegates[delegator];
1037        uint256 delegatorBalance = balanceOf(delegator); //
    ↪  balance of underlying CAKEs (not scaled);
1038        _delegates[delegator] = delegatee;
1039
1040        emit DelegateChanged(delegator, currentDelegate,
    ↪  delegatee);
1041
1042        _moveDelegates(currentDelegate, delegatee,
    ↪  delegatorBalance);
1043    }
1044
1045    function _moveDelegates(address srcRep, address dstRep,
    ↪  uint256 amount) internal {
1046        if (srcRep != dstRep && amount > 0) {
1047            if (srcRep != address(0)) {
1048                // decrease old representative
1049                uint32 srcRepNum = numCheckpoints[srcRep];
1050                uint256 srcRepOld = srcRepNum > 0 ?
    ↪  checkpoints[srcRep][srcRepNum - 1].votes : 0;
1051                uint256 srcRepNew = srcRepOld.sub(amount);
1052                _writeCheckpoint(srcRep, srcRepNum, srcRepOld,
    ↪  srcRepNew);
```

```
1053            }
1054
1055            if (dstRep != address(0)) {
1056                // increase new representative
1057                uint32 dstRepNum = numCheckpoints[dstRep];
1058                uint256 dstRepOld = dstRepNum > 0 ?
    ↪ checkpoints[dstRep][dstRepNum - 1].votes : 0;
1059                uint256 dstRepNew = dstRepOld.add(amount);
1060                _writeCheckpoint(dstRep, dstRepNum, dstRepOld,
    ↪ dstRepNew);
1061            }
1062        }
1063    }
1064
1065    function _writeCheckpoint(
1066        address delegatee,
1067        uint32 nCheckpoints,
1068        uint256 oldVotes,
1069        uint256 newVotes
1070    )
1071        internal
1072    {
1073        uint32 blockNumber = safe32(block.number,
    ↪ "SAGE::_writeCheckpoint: block number exceeds 32 bits");
1074
1075        if (nCheckpoints > 0 &&
    ↪ checkpoints[delegatee][nCheckpoints - 1].fromBlock ==
    ↪ blockNumber) {
1076            checkpoints[delegatee][nCheckpoints - 1].votes =
    ↪ newVotes;
1077        } else {
1078            checkpoints[delegatee][nCheckpoints] =
    ↪ Checkpoint(blockNumber, newVotes);
1079            numCheckpoints[delegatee] = nCheckpoints + 1;
1080        }
1081
1082        emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
1083    }
1084
1085    function safe32(uint n, string memory errorMessage) internal
    ↪ pure returns (uint32) {
1086        require(n < 2**32, errorMessage);
1087        return uint32(n);
1088    }
1089
1090    function getChainId() internal pure returns (uint) {
```

```
1091            uint256 chainId;
1092            assembly { chainId := chainid() }
1093            return chainId;
1094        }
1095    }
1096
1097    contract MasterChef is Ownable { //←
1098        using SafeMath for uint256;
1099        using SafeBEP20 for IBEP20; //←
1100
1101        // Info of each user.
1102        struct UserInfo {
1103            uint256 amount;         // How many LP tokens the user
    ↪   has provided.
1104            uint256 rewardDebt;     // Reward debt. See explanation
    ↪   below.
1105            //
1106            // We do some fancy math here. Basically, any point in
    ↪   time, the amount of sages
1107            // entitled to a user but is pending to be distributed
    ↪   is:
1108            //
1109            //   pending reward = (user.amount *
    ↪   pool.accSagePerShare) - user.rewardDebt
1110            //
1111            // Whenever a user deposits or withdraws LP tokens to a
    ↪   pool. Here's what happens:
1112            //   1. The pool's `accSagePerShare` (and
    ↪   `lastRewardBlock`) gets updated.
1113            //   2. User receives the pending reward sent to his/her
    ↪   address.
1114            //   3. User's `amount` gets updated.
1115            //   4. User's `rewardDebt` gets updated. Try Again
1116        }
1117
1118        // Info of each pool.
1119        struct PoolInfo {
1120            IBEP20 lpToken;           // Address of LP token
    ↪   contract. ←
1121            uint256 allocPoint;       // How many allocation points
    ↪   assigned to this pool. sages to distribute per block.
1122            uint256 lastRewardBlock;  // Last block number that sages
    ↪   distribution occurs.
1123            uint256 accSagePerShare;   // Accumulated SAGEs per
    ↪   share, times 1e12. See below.
1124            uint16 depositFeeBP;      // Deposit fee in basis points
```

```
1125     }
1126
1127     // The SAGE TOKEN!!
1128     SageToken public sage; //←
1129     // Dev address.
1130     address public devaddr;
1131     // SAGE tokens created per block.
1132     uint256 public sagePerBlock;
1133     // Bonus muliplier for early sage makers.
1134     uint256 public constant BONUS_MULTIPLIER = 1;
1135     // Deposit Fee address.
1136     address public feeAddress;
1137     // Owner address.
1138     address public owner;
1139
1140         // Initial emission rate: 1 SAIL per block.
1141     uint256 public constant INITIAL_EMISSION_RATE = 1 ether;
1142     // Minimum emission rate: 0.1 SAIL per block.
1143     uint256 public constant MINIMUM_EMISSION_RATE = 100 finney;
1144     // Reduce emission every 7,200 blocks ~ 6 hours.
1145     uint256 public constant EMISSION_REDUCTION_PERIOD_BLOCKS =
     ↪ 7200;
1146     // Emission reduction rate per period in basis points: 3%.
1147     uint256 public constant EMISSION_REDUCTION_RATE_PER_PERIOD =
     ↪ 300;
1148     // Last reduction period index
1149     uint256 public lastReductionPeriodIndex = 0;
1150
1151     // Info of each pool.
1152     PoolInfo[] public poolInfo; //←
1153     // Info of each user that stakes LP tokens.
1154     mapping (uint256 => mapping (address => UserInfo)) public
     ↪ userInfo;
1155     // Total allocation points. Must be the sum of all allocation
     ↪ points in all pools.
1156     uint256 public totalAllocPoint = 0;
1157     // The block number when SAGE mining starts.
1158     uint256 public startBlock;
1159
1160     mapping(address => address) public referrers; //
     ↪ account_address -> referrer_address
1161     mapping(address => uint256) public referredCount; //
     ↪ referrer_address -> num_of_referred
1162
1163     event Referral(address indexed referrer, address indexed
     ↪ farmer);
```

```
1164      event ReferralPaid(address indexed user,address indexed
      ↪  userTo, uint256 reward);
1165
1166      event Deposit(address indexed user, uint256 indexed pid,
      ↪  uint256 amount);
1167      event Withdraw(address indexed user, uint256 indexed pid,
      ↪  uint256 amount);
1168      event EmergencyWithdraw(address indexed user, uint256 indexed
      ↪  pid, uint256 amount);
1169      event EmissionRateUpdated(address indexed caller, uint256
      ↪  previousAmount, uint256 newAmount);
1170
1171      constructor(
1172          SageToken _sage,
1173          address _feeAddress,
1174          address _owner,
1175          uint256 _startBlock
1176      ) public {
1177          sage = _sage;
1178          devaddr = msg.sender;
1179          owner = _owner;
1180          feeAddress = _feeAddress;
1181          sagePerBlock = INITIAL_EMISSION_RATE;
1182          startBlock = _startBlock;
1183      }
1184
1185      function poolLength() external view returns (uint256) {
1186          return poolInfo.length;
1187      }
1188
1189      // Add a new lp to the pool. Can only be called by the owner.
1190      // XXX DO NOT add the same LP token more than once. Rewards
      ↪  will be messed up if you do.
1191      function add(uint256 _allocPoint, IBEP20 _lpToken, uint16
      ↪  _depositFeeBP, bool _withUpdate) public onlyOwner {//←
1192          require(_depositFeeBP <= 10000, "add: invalid deposit fee
      ↪  basis points");
1193          if (_withUpdate) {
1194              massUpdatePools();
1195          }
1196          uint256 lastRewardBlock = block.number > startBlock ?
      ↪  block.number : startBlock;
1197          totalAllocPoint = totalAllocPoint.add(_allocPoint);
1198          poolInfo.push(PoolInfo({
1199              lpToken: _lpToken,
1200              allocPoint: _allocPoint,
```

```
1201            lastRewardBlock: lastRewardBlock,
1202            accSagePerShare: 0,
1203            depositFeeBP: _depositFeeBP
1204        }));
1205    }
1206
1207    // Update the given pool's SAGE allocation point and deposit
    ↪ fee. Can only be called by the owner.
1208    function set(uint256 _pid, uint256 _allocPoint, uint16
    ↪ _depositFeeBP, bool _withUpdate) public onlyOwner {
1209        require(_depositFeeBP <= 10000, "set: invalid deposit fee
    ↪ basis points");
1210        if (_withUpdate) {
1211            massUpdatePools();
1212        }
1213        totalAllocPoint =
    ↪ totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
1214        poolInfo[_pid].allocPoint = _allocPoint;
1215        poolInfo[_pid].depositFeeBP = _depositFeeBP;
1216    }
1217
1218    // Return reward multiplier over the given _from to _to
    ↪ block.
1219    function getMultiplier(uint256 _from, uint256 _to) public
    ↪ view returns (uint256) {
1220        return _to.sub(_from).mul(BONUS_MULTIPLIER);
1221    }
1222
1223    // View function to see pending SAGEs on frontend.
1224    function pendingSage(uint256 _pid, address _user) external
    ↪ view returns (uint256) {
1225        PoolInfo storage pool = poolInfo[_pid];
1226        UserInfo storage user = userInfo[_pid][_user];
1227        uint256 accSagePerShare = pool.accSagePerShare;
1228        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1229        if (block.number > pool.lastRewardBlock && lpSupply != 0)
    ↪ {
1230            uint256 multiplier =
    ↪ getMultiplier(pool.lastRewardBlock, block.number);
1231            uint256 sageReward =
    ↪ multiplier.mul(sagePerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1232            accSagePerShare =
    ↪ accSagePerShare.add(sageReward.mul(1e12).div(lpSupply));
1233        }
1234        return
    ↪ user.amount.mul(accSagePerShare).div(1e12).sub(user.rewardDebt);
```

```
1235        }
1236
1237        // Update reward variables for all pools. Be careful of gas
    ↪   spending!
1238        function massUpdatePools() public {
1239            uint256 length = poolInfo.length;
1240            for (uint256 pid = 0; pid < length; ++pid) {
1241                updatePool(pid);
1242            }
1243        }
1244
1245        // Update reward variables of the given pool to be
    ↪   up-to-date.
1246        function updatePool(uint256 _pid) public {
1247            PoolInfo storage pool = poolInfo[_pid];
1248            if (block.number <= pool.lastRewardBlock) {
1249                return;
1250            }
1251            uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1252            if (lpSupply == 0 || pool.allocPoint == 0) {
1253                pool.lastRewardBlock = block.number;
1254                return;
1255            }
1256            uint256 multiplier = getMultiplier(pool.lastRewardBlock,
    ↪   block.number);
1257            uint256 sageReward =
    ↪   multiplier.mul(sagePerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1258            sage.mint(address(this), sageReward); //←
1259            pool.accSagePerShare =
    ↪   pool.accSagePerShare.add(sageReward.mul(1e12).div(lpSupply));
1260            pool.lastRewardBlock = block.number;
1261        }
1262
1263        // Deposit LP tokens to MasterChef for SAGE allocation.
1264        function deposit(uint256 _pid, uint256 _amount,address
    ↪   referrer) public   {
1265            PoolInfo storage pool = poolInfo[_pid];
1266            UserInfo storage user = userInfo[_pid][msg.sender];
1267            updatePool(_pid);
1268             if (_amount>0 && referrer != address(0)) {
1269                setRefFriend(msg.sender, referrer);
1270            }
1271            if (user.amount > 0) {
1272                uint256 pending =
    ↪   user.amount.mul(pool.accSagePerShare).div(1e12).sub(user.rewardDebt);
```

```
1273            uint256 toReferral = pending.mul(3).div(100); //
    ↪ percent
1274
1275            if(pending > 0) {
1276               referrer = getRefFriend(msg.sender);
1277               if (referrer != address(0)) { // send commission
    ↪ to referrer
1278                  sage.mint(referrer, toReferral); //←
1279                  emit ReferralPaid(msg.sender,
    ↪ referrer,toReferral);
1280               }
1281
1282               safeSageTransfer(msg.sender, pending);
1283            }
1284        }
1285        if(_amount > 0) {
1286            pool.lpToken.safeTransferFrom(address(msg.sender),
    ↪ address(this), _amount);
1287               if (address(pool.lpToken) == address(sage)) {
1288               uint256 transferTax = _amount.mul(2).div(100);
1289               _amount = _amount.sub(transferTax);
1290            }
1291            if(pool.depositFeeBP > 0){
1292               uint256 depositFee =
    ↪ _amount.mul(pool.depositFeeBP).div(10000);
1293               pool.lpToken.safeTransfer(feeAddress,
    ↪ depositFee);
1294               user.amount =
    ↪ user.amount.add(_amount).sub(depositFee);
1295            }else{
1296               user.amount = user.amount.add(_amount);
1297            }
1298        }
1299        user.rewardDebt =
    ↪ user.amount.mul(pool.accSagePerShare).div(1e12);
1300        emit Deposit(msg.sender, _pid, _amount);
1301    }
1302
1303    // Withdraw LP tokens from MasterChef.
1304    function withdraw(uint256 _pid, uint256 _amount) public {
1305        PoolInfo storage pool = poolInfo[_pid];
1306        UserInfo storage user = userInfo[_pid][msg.sender];
1307        require(user.amount >= _amount, "withdraw: not good");
1308        updatePool(_pid);
1309        uint256 pending =
    ↪ user.amount.mul(pool.accSagePerShare).div(1e12).sub(user.rewardDebt);
```

```
1310            if(pending > 0) {
1311                safeSageTransfer(msg.sender, pending);
1312            }
1313            if(_amount > 0) {
1314                user.amount = user.amount.sub(_amount);
1315                pool.lpToken.safeTransfer(address(msg.sender),
    ↪  _amount);
1316            }
1317            user.rewardDebt =
    ↪  user.amount.mul(pool.accSagePerShare).div(1e12);
1318            emit Withdraw(msg.sender, _pid, _amount);
1319        }
1320
1321        // Withdraw without caring about rewards. EMERGENCY ONLY.
1322        function emergencyWithdraw(uint256 _pid) public {
1323            PoolInfo storage pool = poolInfo[_pid];
1324            UserInfo storage user = userInfo[_pid][msg.sender];
1325            uint256 amount = user.amount;
1326            user.amount = 0;
1327            user.rewardDebt = 0;
1328            pool.lpToken.safeTransfer(address(msg.sender), amount);
1329            emit EmergencyWithdraw(msg.sender, _pid, amount);
1330        }
1331
1332        // Safe sage transfer function, just in case if rounding
    ↪  error causes pool to not have enough SAGEs.
1333        function safeSageTransfer(address _to, uint256 _amount)
    ↪  internal {
1334            uint256 sageBal = sage.balanceOf(address(this));
1335            bool transferSuccess = false;
1336            if (_amount > sageBal) {
1337                transferSuccess = sage.transfer(_to, sageBal);
1338            } else {
1339                transferSuccess = sage.transfer(_to, _amount);
1340            }
1341                    require(transferSuccess, "safeSageTransfer:
    ↪  Transfer failed");
1342        }
1343
1344        // Update dev address by the previous dev.
1345        function dev(address _devaddr) public {
1346            require(msg.sender == devaddr, "dev: wut?");
1347            devaddr = _devaddr;
1348        }
1349
1350        function setFeeAddress(address _feeAddress) public {
```

```
1351        require(msg.sender == feeAddress, "setFeeAddress:
    ↪ FORBIDDEN");
1352        feeAddress = _feeAddress;
1353    }
1354
1355    // Reduce emission rate by 3% every 9,600 blocks ~ 8hours.
    ↪ This function can be called publicly.
1356    function updateEmissionRate() public {
1357        require(block.number > startBlock, "updateEmissionRate:
    ↪ Can only be called after mining starts");
1358        require(sagePerBlock > MINIMUM_EMISSION_RATE,
    ↪ "updateEmissionRate: Emission rate has reached the minimum
    ↪ threshold");
1359        uint256 currentIndex =
    ↪ block.number.sub(startBlock).div(EMISSION_REDUCTION_PERIOD_BLOCKS);
1360        if (currentIndex <= lastReductionPeriodIndex) {
1361            return;
1362        }
1363        uint256 newEmissionRate = sagePerBlock;
1364        for (uint256 index = lastReductionPeriodIndex; index <
    ↪ currentIndex; ++index) {
1365            newEmissionRate = newEmissionRate.mul(1e4 -
    ↪ EMISSION_REDUCTION_RATE_PER_PERIOD).div(1e4);
1366        }
1367        newEmissionRate = newEmissionRate < MINIMUM_EMISSION_RATE
    ↪ ? MINIMUM_EMISSION_RATE : newEmissionRate;
1368        if (newEmissionRate >= sagePerBlock) {
1369            return;
1370        }
1371        massUpdatePools();
1372        lastReductionPeriodIndex = currentIndex;
1373        uint256 previousEmissionRate = sagePerBlock;
1374        sagePerBlock = newEmissionRate;
1375        emit EmissionRateUpdated(msg.sender,
    ↪ previousEmissionRate, newEmissionRate);
1376    }
1377
1378    function setRefFriend(address farmer, address referrer)
    ↪ internal {
1379        if (referrers[farmer] == address(0) && referrer !=
    ↪ address(0)) {
1380            referrers[farmer] = referrer;
1381            referredCount[referrer] += 1;
1382            emit Referral(referrer, farmer);
1383        }
1384    }
```

```
1385
1386        function getRefFriend(address farmer) public view returns
    ↪   (address) {
1387            return referrers[farmer];
1388        }
1389
1390    }
```