Audit Report

Produced by CertiK

for bZx

Additional Formal Verification Coverage of BZRX ERC Token and Vesting Mechanism
CertiK Audit Report
For BzX

Request Date: 2020-07-07
Revision Date: 2020-07-11
Platform Name: Ethereum
Contents

Disclaimer .................................................. 1
About CertiK ................................................. 2
Executive Summary ...................................... 3
Vulnerability Classification ............................ 3
Testing Summary ........................................ 4
  Audit Score ............................................. 4
  Type of Issues ......................................... 4
  Vulnerability Details .................................. 5
Manual Review Notes ................................... 6
Static Analysis Results ................................. 7
Formal Verification Results ............................ 8
  How to read ............................................. 8
Source Code with CertiK Labels ..................... 17
Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and BzX (the “Company”), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the “Agreement”). This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK’s prior written consent.
About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, has developed a proprietary Formal Verification technology to apply rigorous and complete mathematical reasoning against code. This process ensures algorithms, protocols, and business functionalities are secured and working as intended across all platforms.

CertiK differs from traditional testing approaches by employing Formal Verification to mathematically prove blockchain ecosystem and smart contracts are hacker-resistant and bug-free. CertiK uses this industry-leading technology together with standardized test suites, static analysis, and expert manual review to create a full-stack solution for our partners across the blockchain world to secure 6.2B in assets.

For more information: https://certik.org/
Executive Summary

This report has been prepared for BzX to discover issues and vulnerabilities in the source code of their smart contracts. A comprehensive examination has been performed, utilizing CertiK’s Formal Verification Platform, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Vulnerability Classification

CertiK categorizes issues into three buckets based on overall risk levels:

**Critical**

Code implementation does not match specification, which could result in the loss of funds for contract owner or users.

**Medium**

Code implementation does not match the specification under certain conditions, which could affect the security standard by loss of access control.

**Low**

Code implementation does not follow best practices, or uses suboptimal design patterns, which could lead to security vulnerabilities further down the line.
Testing Summary

CERTIK believes this smart contract passes security qualifications to be listed on digital asset exchanges.

Jul 11, 2020

Type of Issues

CertiK’s smart label engine applied 100% formal verification coverage on the source code. Our team of engineers has scanned the source code using proprietary static analysis tools and code-review methodologies. The following technical issues were found:

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Issues</th>
<th>SWC ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Overflow/Underflow</td>
<td>An overflow/underflow occurs when an arithmetic operation reaches the maximum or minimum size of a type.</td>
<td>0</td>
<td>SWC-101</td>
</tr>
<tr>
<td>Function Incorrectness</td>
<td>Function implementation does not meet specification, leading to intentional or unintentional vulnerabilities.</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>An attacker can write to arbitrary storage locations of a contract if array of out bound happens</td>
<td>0</td>
<td>SWC-124</td>
</tr>
<tr>
<td>Reentrancy</td>
<td>A malicious contract can call back into the calling contract before the first invocation of the function is finished.</td>
<td>0</td>
<td>SWC-107</td>
</tr>
<tr>
<td>Transaction Order Dependence</td>
<td>A race condition vulnerability occurs when code depends on the order of the transactions submitted to it.</td>
<td>0</td>
<td>SWC-114</td>
</tr>
<tr>
<td>Timestamp Dependence</td>
<td>Timestamp can be influenced by miners to some degree.</td>
<td>0</td>
<td>SWC-116</td>
</tr>
<tr>
<td>Insecure Randomness</td>
<td>Using block attributes to generate random numbers is unreliable, as they can be influenced by miners to some degree.</td>
<td>0</td>
<td>SWC-120</td>
</tr>
<tr>
<td>“tx.origin” for Authorization</td>
<td>tx.origin should not be used for authorization. Use msg.sender instead.</td>
<td>0</td>
<td>SWC-115</td>
</tr>
<tr>
<td>Delegatecall to Untrusted Callee</td>
<td>Calling untrusted contracts is very dangerous, so the target and arguments provided must be sanitized.</td>
<td>0</td>
<td>SWC-112</td>
</tr>
<tr>
<td>Title</td>
<td>Description</td>
<td>Issues</td>
<td>SWC ID</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>State Variable Default Visibility</td>
<td>Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.</td>
<td>0</td>
<td>SWC-108</td>
</tr>
<tr>
<td>Function Default Visibility</td>
<td>Functions are public by default, meaning a malicious user can make unauthorized or unintended state changes if a developer forgot to set the visibility.</td>
<td>0</td>
<td>SWC-100</td>
</tr>
<tr>
<td>Uninitialized Variables</td>
<td>Uninitialized local storage variables can point to other unexpected storage variables in the contract.</td>
<td>0</td>
<td>SWC-109</td>
</tr>
<tr>
<td>Assertion Failure</td>
<td>The assert() function is meant to assert invariants. Properly functioning code should never reach a failing assert statement.</td>
<td>0</td>
<td>SWC-110</td>
</tr>
<tr>
<td>Deprecated Solidity Features</td>
<td>Several functions and operators in Solidity are deprecated and should not be used.</td>
<td>0</td>
<td>SWC-111</td>
</tr>
<tr>
<td>Unused Variables</td>
<td>Unused variables reduce code quality</td>
<td>0</td>
<td>SWC-131</td>
</tr>
</tbody>
</table>

**Vulnerability Details**

- **Critical**
  - No issue found.

- **Medium**
  - No issue found.

- **Low**
  - Issue 1:
    - Issue 1 *code*.
    - Issue 1 *emphasis*. 
Manual Review Notes

Source Code SHA-256

- Checkpointing.sol
- CheckpointingToken.sol
- BZRXTokken.sol
- BZRXVestingToken.sol

commit hash cc39fc93d05bbeade8bea1d191870c3a4f879ea2. All four smart contracts were manually reviewed in an independent report. Additionally Checkpointing.sol (except for _getValueAt with the while loop) was formally verified and the results can be found in this report.

Summary

The codebase of the project is relatively straightforward and the additional functionality that was introduced with regards to a checkpointed balance sheet was implemented based on source code from Aragon One, an established entity of the Ethereum developer community. CertiK worked closely with bZx to audit the design and implementation of its soon-to-be released smart contract. To ensure comprehensive protection, the source code was analyzed by the proprietary CertiK formal verification engine and manually reviewed by our smart contract experts and engineers. That end-to-end process ensures proof of stability as well as a hands-on, engineering-focused process to close potential loopholes and recommend design changes in accordance with best practices.

Overall, we found bZx’s smart contracts to follow good practices. With the final update of source code and delivery of the audit report, we conclude that the contract is structurally sound and not vulnerable to any classically known anti-patterns or security issues. The audit report itself is not necessarily a guarantee of correctness or trustworthiness, and we always recommend to seek multiple opinions, continually improve the codebase, and perform additional tests before the mainnet release.

Recommendations

Overall, the codebase of the contracts should be refactored to assimilate the findings of this report, enforce linters and / or coding styles as well as correct any spelling errors and mistakes that appear throughout the code to achieve a high standard of code quality and security.
Static Analysis Results

INSECURE_COMPILER_VERSION

Line 6 in File Checkpointing.sol

```
pragma solidity 0.5.17;
```

⚠️ Version to compile has the following bug: 0.5.17: TupleAssignmentMultiStackSlotComponents, MemoryArrayCreationOverflow
Formal Verification Results

How to read

Detail for Request 1

transferFrom to same address

<table>
<thead>
<tr>
<th>Verification date</th>
<th>20, Oct 2018</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification timespan</td>
<td>395.38 ms</td>
</tr>
</tbody>
</table>

CERTIK label location

Line 30-34 in File howtoread.sol

CERTIK label

30 /*CTK FAIL "transferFrom to same address"
31 @tag assume_completion
32 @pre from == to
33 @post __post.allowed[from][msg.sender] ==
34 */

Raw code location

Line 35-41 in File howtoread.sol

Raw code

35 function transferFrom(address from, address to)
36     { balances[from] = balances[from].sub(tokens)
37     allowed[from][msg.sender] = allowed[from][
38     balances[to] = balances[to].add(tokens);
39     emit Transfer(from, to, tokens);
40     return true;
41     }

Counterexample

This code violates the specification

Initial environment

1 Counter Example:
2 Before Execution:
3     Input = {
4         from = 0x0
5         to = 0x0
6         tokens = 0x6c
7     }
8     This = 0

53     balance: 0x0
54     }
55     }
56

After Execution:
57     Input = {
58         from = 0x0
59         to = 0x0
60         tokens = 0x6c
61     }
Formal Verification Request 1

addCheckpoint_empty_history

11, Jul 2020
42.44 ms

Line 40-44 in File Checkpointing.sol

40    /*@CTK addCheckpoint_empty_history
41    @pre _self.history.length == 0
42    @post _self._post.history.length == 1
43    @post _self._post.history[0].time == _time && _self._post.history[0].value == _value
44    */

Line 64-85 in File Checkpointing.sol

64    function addCheckpoint(
65    History storage _self,
66    uint256 _time,
67    uint256 _value
68    ) internal
69    {
70    uint256 length = _self.history.length;
71    if (length == 0) {
72    _self.history.push(Checkpoint(_time, _value));
73    } else {
74    Checkpoint storage currentCheckpoint = _self.history[length - 1];
75    uint256 currentCheckpointTime = currentCheckpoint.time;
76
77    if (_time > currentCheckpointTime) {
78    _self.history.push(Checkpoint(_time, _value));
79    } else if (_time == currentCheckpointTime) {
80    currentCheckpoint.value = _value;
81    } else { // ensure list ordering
82    revert("past-checkpoint");
83    }
84    }
85    }

The code meets the specification.

Formal Verification Request 2

addCheckpoint_non_empty_history_new_data

11, Jul 2020
78.7 ms

Line 45-51 in File Checkpointing.sol

45    /*@CTK addCheckpoint_non_empty_history_new_data
46    @pre _self.history.length > 0
47    @pre _self.history[_self.history.length - 1].time < _time
48    @post _self._post.history.length == _self.history.length + 1
49    @post _self._post.history[_self._post.history.length - 1].time == _time
50    @post _self._post.history[_self._post.history.length - 1].value == _value
51    */
Line 64-85 in File Checkpointing.sol

```solidity
function addCheckpoint(
    History storage _self,
    uint256 _time,
    uint256 _value
) internal
{
    uint256 length = _self.history.length;
    if (length == 0) {
        _self.history.push(Checkpoint(_time, _value));
    } else {
        Checkpoint storage currentCheckpoint = _self.history[length - 1];
        uint256 currentCheckpointTime = currentCheckpoint.time;
        if (_time > currentCheckpointTime) {
            _self.history.push(Checkpoint(_time, _value));
        } else if (_time == currentCheckpointTime) {
            currentCheckpoint.value = _value;
        } else { // ensure list ordering
            revert("past-checkpoint");
        }
    }
}
```

The code meets the specification.

Formal Verification Request 3

addCheckpoint_non_empty_history_current_data

📅 11, Jul 2020
⏰ 0.36 ms

Line 52-58 in File Checkpointing.sol

```solidity
/*@CTK addCheckpoint_non_empty_history_current_data */
@pre _self.history.length > 0
@pre _self.history[_self.history.length - 1].time == _time
@inv _self.history.length
@inv _self.history[_self.history.length - 1].time
@inv _self._post.history[_self._post.history.length - 1].value == _value
```

Line 64-85 in File Checkpointing.sol

```solidity
function addCheckpoint(
    History storage _self,
    uint256 _time,
    uint256 _value
) internal
{
    uint256 length = _self.history.length;
    if (length == 0) {
        _self.history.push(Checkpoint(_time, _value));
    } else {
        Checkpoint storage currentCheckpoint = _self.history[length - 1];
        uint256 currentCheckpointTime = currentCheckpoint.time;
```
if (_time > currentCheckpointTime) {
    _self.history.push(Checkpoint(_time, _value));
} else if (_time == currentCheckpointTime) {
    currentCheckpoint.value = _value;
} else { // ensure list ordering
    revert("past-checkpoint");
}

✓ The code meets the specification.

Formal Verification Request 4

addCheckpoint__non_empty__history__past_data

📅 11, Jul 2020
⏰ 0.33 ms

Line 59-63 in File Checkpointing.sol

```solidity
/*@CTK addCheckpoint_non_empty_history_past_data
@pre _self.history.length > 0
@pre _self.history[_self.history.length - 1].time > _time
@inv _self
*/

function addCheckpoint(
    History storage _self,
    uint256 _time,
    uint256 _value)
internal
{
    uint256 length = _self.history.length;
    if (length == 0) {
        _self.history.push(Checkpoint(_time, _value));
    } else {
        Checkpoint storage currentCheckpoint = _self.history[length - 1];
        uint256 currentCheckpointTime = currentCheckpoint.time;

        if (_time > currentCheckpointTime) {
            _self.history.push(Checkpoint(_time, _value));
        } else if (_time == currentCheckpointTime) {
            currentCheckpoint.value = _value;
        } else { // ensure list ordering
            revert("past-checkpoint");
        }
    }
}
```

✓ The code meets the specification.
Formal Verification Request 5

lastUpdated_zero_history

11, Jul 2020
11.34 ms

Line 97-100 in File Checkpointing.sol

```solidity
97    /*!@CTK lastUpdated_zero_history
98    @pre _self.history.length == 0
99    @post __return == 0
100   */
105    function lastUpdated(
106        History storage _self)
107        internal
108        view
109        returns (uint256)
110    {
111        uint256 length = _self.history.length;
112        if (length != 0) {
113            return _self.history[length - 1].time;
114        }
115        return 0;
116    }
```

✔️ The code meets the specification.

Formal Verification Request 6

latestValue_zero_history

11, Jul 2020
10.58 ms

Line 119-122 in File Checkpointing.sol

```solidity
119    /*!@CTK latestValue_zero_history
120    @pre _self.history.length == 0
121    @post __return == 0
122   */
127    function latestValue(
128        History storage _self)
129        internal
130        view
131        returns (uint256)
132    {
133        uint256 length = _self.history.length;
134        if (length != 0) {
135            return _self.history[length - 1].value;
136        }
137```

137
```
  return 0;
}

✓ The code meets the specification.

**Formal Verification Request 7**

`__getValueAt__empty__history`

📅 11, Jul 2020
🕒 24.66 ms

Line 141-144 in File Checkpointing.sol

```solidity
/*@CTK __getValueAt__empty__history*/
@pre __self__.history.length == 0
@post __return__ == 0
*/
```

Line 155-207 in File Checkpointing.sol

```solidity
function __getValueAt__ (  
    History storage __self__,  
    uint256 __time__)  
private  
view  
returns (uint256)  
{
    uint256 length = __self__.history.length;
    // Short circuit if there's no checkpoints yet
    // Note that this also lets us avoid using SafeMath later on, as we've established that
    // there must be at least one checkpoint
    if (length == 0) {
        return 0;
    }
    // Check last checkpoint
    uint256 lastIndex = length - 1;
    Checkpoint storage lastCheckpoint = __self__.history[lastIndex];
    if (__time >= lastCheckpoint.time) {
        return lastCheckpoint.value;
    }
    // Check first checkpoint (if not already checked with the above check on last)
    if (length == 1 || __time < __self__.history[0].time) {
        return 0;
    }
    // Do binary search
    // As we've already checked both ends, we don't need to check the last checkpoint again
    uint256 low = 0;
    uint256 high = lastIndex - 1;
    /* while (high != low) { *
```
uint256 mid = (high + low + 1) / 2; // average, ceil round
Checkpoint storage checkpoint = _self.history[mid];
uint256 midTime = checkpoint.time;

if (_time > midTime) {
    low = mid;
} else if (_time < midTime) {
    // Note that we don’t need SafeMath here because mid must always be greater
    // than 0
    // from the while condition
    high = mid - 1;
} else {
    // _time == midTime
    return checkpoint.value;
}

return _self.history[low].value;

✓ The code meets the specification.

Formal Verification Request 8
_getValueAt_current_checkpoint

📅 11, Jul 2020
⏰ 5.82 ms

Line 145-149 in File Checkpointing.sol

```solidity
/*@CTK _getValueAt_current_checkpoint
@pre _self.history.length > 0
@pre _self.history[_self.history.length - 1].time <= _time
@post __return == _self.history[_self.history.length - 1].value
*/
```

Line 155-207 in File Checkpointing.sol

```solidity
function _getValueAt(
    History storage _self,
    uint256 _time)
private
view
returns (uint256)
{
    uint256 length = _self.history.length;
    // Short circuit if there's no checkpoints yet
    // Note that this also lets us avoid using SafeMath later on, as we've established
    // there must be at least one checkpoint
    if (length == 0) {
        return 0;
    }
    // Check last checkpoint
    uint256 lastIndex = length - 1;
```
```solidity
    Checkpoint storage lastCheckpoint = _self.history[lastIndex];
    if (_time >= lastCheckpoint.time) {
        return lastCheckpoint.value;
    }
    // Check first checkpoint (if not already checked with the above check on last)
    if (length == 1 || _time < _self.history[0].time) {
        return 0;
    }
    // Do binary search
    // As we’ve already checked both ends, we don’t need to check the last checkpoint again
    uint256 low = 0;
    uint256 high = lastIndex - 1;
    while (high != low) {
        uint256 mid = (high + low + 1) / 2; // average, ceil round
        Checkpoint storage checkpoint = _self.history[mid];
        uint256 midTime = checkpoint.time;
        if (_time > midTime) {
            low = mid;
        } else if (_time < midTime) {
            // Note that we don’t need SafeMath here because mid must always be greater than 0
            high = mid - 1;
        } else {
            // _time == midTime
            return checkpoint.value;
        }
    }
    return _self.history[low].value;
```

The code meets the specification.

**Formal Verification Request 9

_getValueAt__past__checkpoint

📅 11, Jul 2020
⏰ 8.51 ms

Line 150-154 in File Checkpointing.sol

```solidity
    /*!CTK _getValueAt__past__checkpoint
    @pre _self.history.length == 1
    @pre _self.history[0].time > _time
    @post __return == 0 */
```

Line 155-207 in File Checkpointing.sol
function _getValueAt(
    History storage _self,
    uint256 _time)
private
view
returns (uint256)
{
    uint256 length = _self.history.length;

    // Short circuit if there's no checkpoints yet
    // Note that this also lets us avoid using SafeMath later on, as we've established
    // there must be at least one checkpoint
    if (length == 0) {
        return 0;
    }

    // Check last checkpoint
    uint256 lastIndex = length - 1;
    Checkpoint storage lastCheckpoint = _self.history[lastIndex];
    if (_time >= lastCheckpoint.time) {
        return lastCheckpoint.value;
    }

    // Check first checkpoint (if not already checked with the above check on last)
    if (length == 1 || _time < _self.history[0].time) {
        return 0;
    }

    // Do binary search
    // As we've already checked both ends, we don't need to check the last checkpoint
    // again
    uint256 low = 0;
    uint256 high = lastIndex - 1;

    /* while (high != low) {
        uint256 mid = (high + low + 1) / 2; // average, ceil round
        Checkpoint storage checkpoint = _self.history[mid];
        uint256 midTime = checkpoint.time;

        if (_time > midTime) {
            low = mid;
        } else if (_time < midTime) {
            // Note that we don't need SafeMath here because mid must always be greater
            // than 0
            high = mid - 1;
        } else {
            // _time == midTime
            return checkpoint.value;
        }
    } */

    return _self.history[low].value;
}  

✅ The code meets the specification.
File Checkpointing.sol

```solidity
pragma solidity 0.5.17;

/*
 * Copyright (C) 2019 Aragon One <https://aragon.one/>
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

/*
 * @title Checkpointing
 *
 * @notice Checkpointing library for keeping track of historical values based on an
 *         arbitrary time unit (e.g. seconds or block numbers).
 * @dev Adapted from:
 *      - Checkpointing (https://github.com/aragonone/voting-connectors/blob/master/shared/
 *        contract-utils/contracts/Checkpointing.sol)
 */

library Checkpointing {

    struct Checkpoint {
        uint256 time;
        uint256 value;
    }

    struct History {
        Checkpoint[] history;
    }

    /*CTK addCheckpoint_empty_history
    @pre _self.history.length == 0
    @post _self._post.history.length == 1
    @post _self._post.history[0].time == _time & _self._post.history[0].value == _value
    */

    /*CTK addCheckpoint_non_empty_history_new_data
    @pre _self.history.length > 0
    @pre _self._self.history[_self.history.length - 1].time < _time
    @post _self._post.history[_self._post.history.length - 1].time == _time
    @post _self._post.history[_self._post.history.length - 1].value == _value
    */

    /*CTK addCheckpoint_non_empty_history_current_data
```
@pre _self.history.length > 0
@pre _self.history[_self.history.length - 1].time == _time
@inv _self.history.length
@inv _self.history[_self.history.length - 1].time
@inv _self.__post.history[_self.__post.history.length - 1].value == _value
/*
 */
/*@CTK addCheckpoint_non_empty_history_past_data
@pre _self.history.length > 0
@pre _self.history[_self.history.length - 1].time > _time
@inv _self
*/
function addCheckpoint(
    History storage _self,
    uint256 _time,
    uint256 _value)
internal
{
    uint256 length = _self.history.length;
    if (length == 0) {
        _self.history.push(Checkpoint(_time, _value));
    } else {
        Checkpoint storage currentCheckpoint = _self.history[length - 1];
        uint256 currentCheckpointTime = currentCheckpoint.time;

        if (_time > currentCheckpointTime) {
            _self.history.push(Checkpoint(_time, _value));
        } else if (_time == currentCheckpointTime) {
            currentCheckpoint.value = _value;
        } else { // ensure list ordering
            revert("past-checkpoint");
        }
    }
}

function getValueAt(
    History storage _self,
    uint256 _time)
internal
view
returns (uint256)
{
    return _getValueAt(_self, _time);
}

/*@CTK lastUpdated_zero_history
@pre _self.history.length == 0
@post __return == 0
*/
/*@CTK lastUpdated_nonzero_history
@pre _self.history.length != 0
@post __return == _self.history[_self.history.length - 1].time
*/
function lastUpdated(
    History storage _self)
internal
view
returns (uint256)
{
```solidity
uint256 length = _self.history.length;
if (length != 0) {
    return _self.history[length - 1].time;
}
return 0;

/*@CTK latestValue_zero_history*/
@pre _self.history.length == 0
@post __return == 0
/*@CTK latestValue_nonzero_history*/
@pre _self.history.length != 0
@post __return == _self.history[_self.history.length - 1].value
/*@CTK _getValueAt_empty_history*/
@pre _self.history.length == 0
@post __return == 0
/*@CTK _getValueAt_current_checkpoint*/
@pre _self.history.length > 0
@pre _self.history[_self.history.length - 1].time <= __time
@post __return == _self.history[_self.history.length - 1].value
/*@CTK _getValueAt_past_checkpoint*/
@pre _self.history.length == 1
@pre _self.history[0].time > __time
@post __return == 0
/*@CTK _getValueAt*/

function _getValueAt(
    History storage _self,
    uint256 _time
)
private
view
returns (uint256)
{
    uint256 length = _self.history.length;
    // Short circuit if there's no checkpoints yet
    // Note that this also lets us avoid using SafeMath later on, as we've established that
    // there must be at least one checkpoint
    if (length == 0) {
```
```
return 0;
}

// Check last checkpoint
uint256 lastIndex = length - 1;
Checkpoint storage lastCheckpoint = _self.history[lastIndex];
if (_time >= lastCheckpoint.time) {
    return lastCheckpoint.value;
}

// Check first checkpoint (if not already checked with the above check on last)
if (length == 1 || _time < _self.history[0].time) {
    return 0;
}

// Do binary search
// As we've already checked both ends, we don't need to check the last checkpoint
again
uint256 low = 0;
uint256 high = lastIndex - 1;

/* while (high != low) {
    uint256 mid = (high + low + 1) / 2; // average, ceil round
    Checkpoint storage checkpoint = _self.history[mid];
    uint256 midTime = checkpoint.time;
    if (_time > midTime) {
        low = mid;
    } else if (_time < midTime) {
        // Note that we don't need SafeMath here because mid must always be greater
        // than 0
        // from the while condition
        high = mid - 1;
    } else {
        // _time == midTime
        return checkpoint.value;
    }
} */

return _self.history[low].value;
```