MONITOR
CHAIN

MonitorChain integration instruction

MonitorChain is a decentralized oracle on Ethereum blockchain whose purpose is to serve current token state. The Ethereum ecosystem currently has more than 550 tokens that are listed on Etherscan and are traded on various exchanges. There are many more tokens that aren't listed on major exchanges but might become traded in the future.

Table of Contents

MonitorChain UI and features

From the MonitorChain perspective, tokens (and their respective token contracts) can be in various states which are currently divided into the following four categories:

0. Good
1. Notice
2. Warning
3. Severe
4. Emergency

The states depend on the detected potential and actual fraudulent behavior within transactions related to the token contract. In order to detect such behavior MonitorChain relies on two components, monitor and tracer. Monitor updates the MoniotorChain smart contract with appropriate error for the token on which such malicious transaction has been detected. Tracker is tracing the "transaction tree" that starts from the malicious transaction destination address and blocks them on the MonitorChain smart contract.

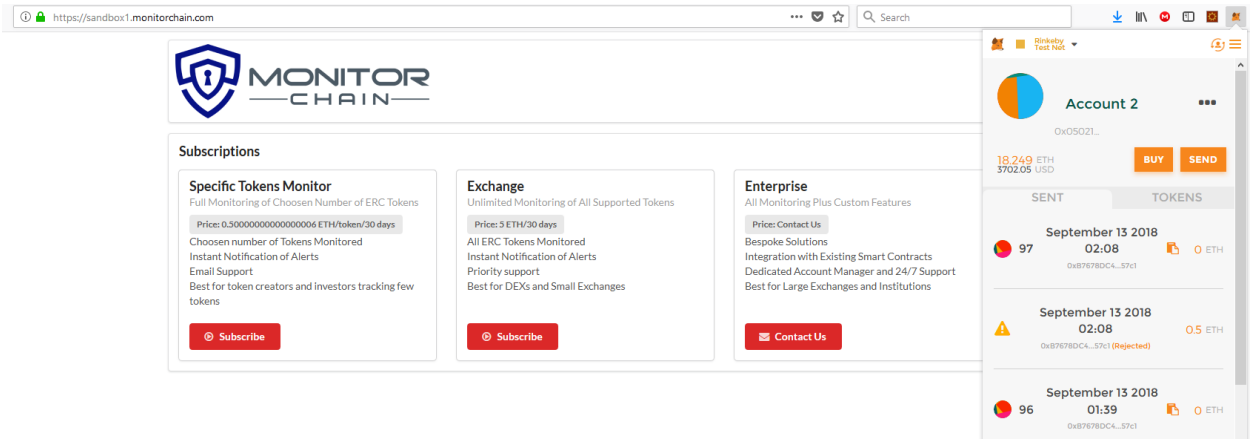

Figure 1 – MonitorChain homepage for unsubscribed user

MonitorChain displays such token statuses on the main dApp UI for subscribers. Depending on the subscription, users can access states of separate tokens or all listed tokens. Subscription is paid in Ethers allows reading the current status of the token but also the token status history. In order to pay for the subscription user navigates with its metamask wallet to the homepage of the MonitorChain (for example: sandbox1.monitorchain.com) (*Figure 1*) and is presented with the page where she can choose her subscription. Also, user can navigate to sandbox1.monitorchain.com/subscriptions (Figure 1) to check her current subscription, extend it or change it. Once subscribed, user can access the token feed as per subscription.

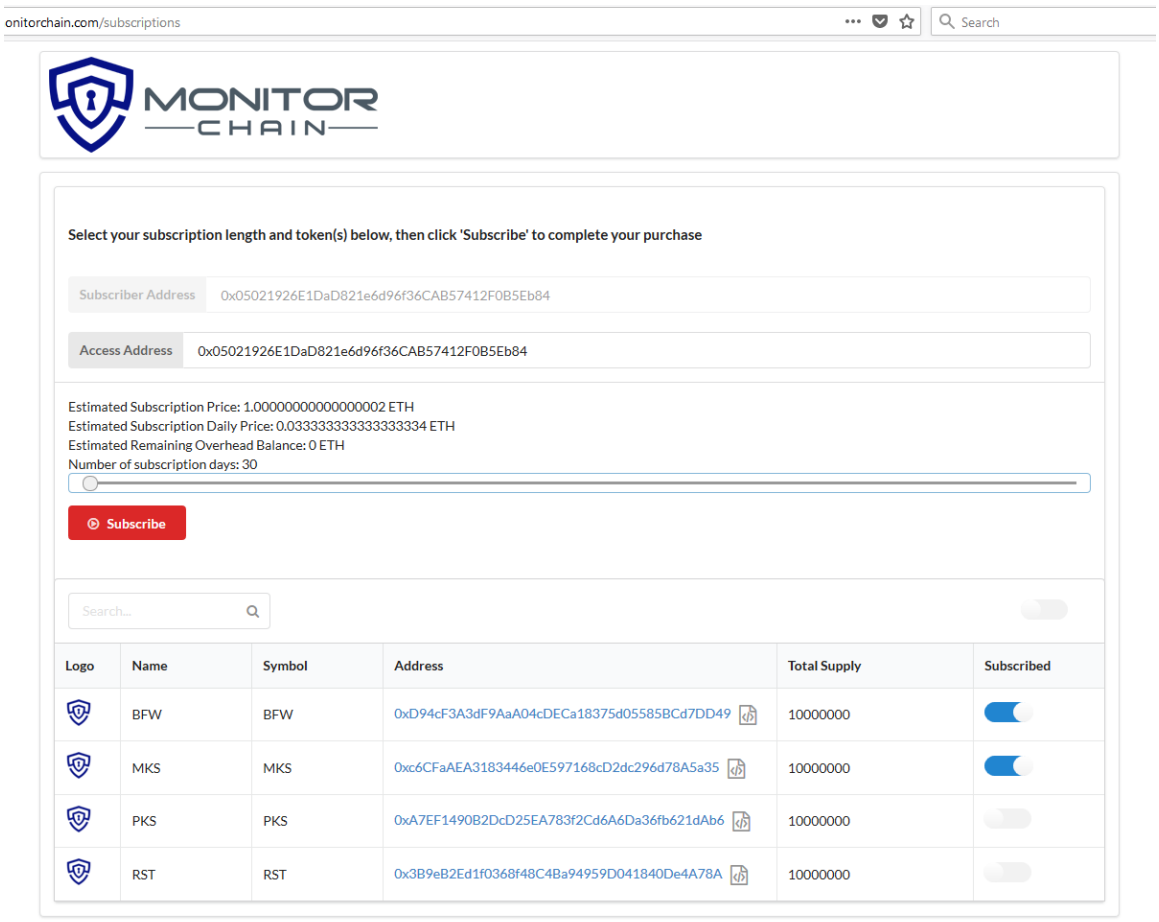

Figure 2 – Subscription page

If user has successfully subscribed to tokens, she can see their statuses in the feed on the MonitorChain homepage (*Figure 3*), but only for those for which the subscription has been paid. From the same page, user can check the details of the current status and navigate to the token status history page.

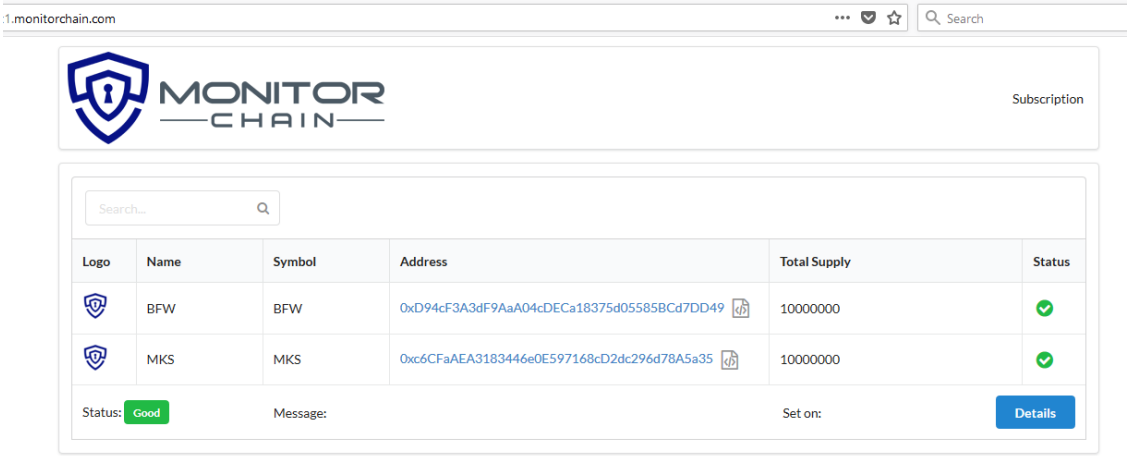

Figure 3 – Homepage token feed

Programmatically Accessing MonitorChain

Besides accessing MonitorChain directly through the token feed of the dApp itself, more common scenario is accessing the MonitorChain programmatically. This feature is important to allow other dApps (exchanges, wallets, token contracts etc.) to be able to read the current token status from the MonitorChain contract and act accordingly. To allow programmatic access to the MonitorChain tokens statuses. user can access MonitorChain subscription page and pay subscription from her own wallet on behalf of another address, called access address. Access address can be an address of another smart contract or a wallet that is used to programmatically access the MonitorChain (*Figure 4*).

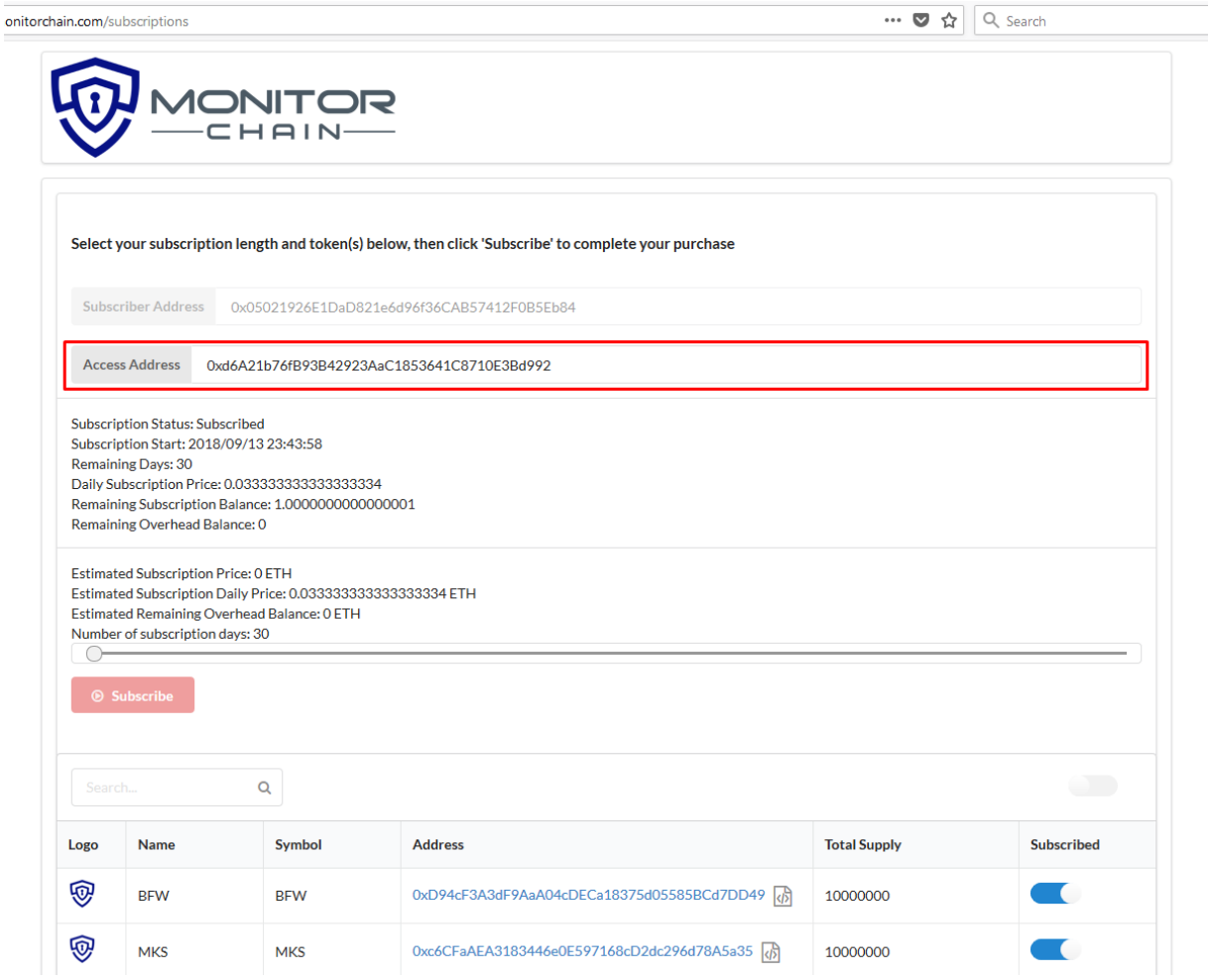

Figure 4 – Setting Access Address for subscription

There are two main approaches for programmatic integration with MonitorChain:

1. Integrating dApp's smart contract with MonitorChain smart contract directly
2. Integrating NodeJS script which is using web3 or similar library to access MonitorChain

Integration of a smart contract directly with MonitorChain smart contract is more suitable for on-chain Dexes that would like to freeze trading in cases when a particular token has error or to block trading for a blocked address.

Integration of a NodeJS script is useful in cases when off chain exchange wants to block trading on a token that has error. Since MonitorChain issues events in cases when token statuses change, NodeJS script tied to a wallet that has access address subscribed to a token which changed, can read its new status and forward such data. For example, such script could block centralized exchange, send email to the trader, send other means of notifications to various interested parties etc. Such script could also affect some other smart contract indirectly submitting a transaction to it in relation to changed token status.

In order to enable developers to test various approaches of integration, MonitorChain provides integration sandboxes. Integration sandboxes contain the following elements that provide developers with the tools for seamless integration:

1. Deployed MonitorChain UI on our servers with URL: sandboxX.monitorchain.com where X is replaced with a number (1,2, etc).
2. MonitorChain smart contract on Rinkeby Ethereum test network with appropriate address of the deployed contract.
3. Four test tokens which are set to be monitored on MonitorChain and which are vulnerable to various malicious transactions. Sandbox version of the MonitorChain contains special page that helps with breaking such tokens.
4. Github repo which contains MonitorChain AccessInterface for NodeJS programmatic access with examples and documentation (https://github.com/ZenchainSoftware/monitorchain-interface-library)
5. Github repo which contains breakable token contract code, smart contract integration pattern examples and MonitorChain reset script (https://github.com/ZenchainSoftware/monitorchain-developer-sandbox-tools)
6. Npm monitorchain-interface-library https://www.npmjs.com/package/monitorchain-interface-library
7. Document with actual sandbox MonitorChain URL, MonitorChain Rinkeby smart contract address, seed phase (mnemonics) of the MonitorChain admin account and account address prefilled with some Rinkeby Ethereum.

AccessInterface contract

For proper integration with MonitorChain, both from smart contract and from NodeJS application, proper contract code that exposes MonitorChain public methods is needed. Such contract is implemented in the AccessInterface contract. The contract can be obtained from two endpoints https://github.com/ZenchainSoftware/monitorchain-interface-library or https://www.npmjs.com/package/monitorchain-interface-library.

The code snippet (*Snippet 1*) gives all the MonitorChain public functions exposed in the AccessLibrary with detail comments

```
contract AccessInterface {
  //returns minimum number of days initial user subscription has to last
  function minDays() public view returns(uint8 minDays);
  //returns price per token per day of the subscription
  function pricePerTokenPerDay() public view returns(uint8 pricePerTokenPerDay);
  //returns price for all tokens per day of the subscription
  function priceForAllPerDay() public view returns(uint8 priceForAllPerDay);
  //for the eventId received from the MonitorChain TokenStatusChanged event, returns the
  //token address if the accessing address is subscribed for the token
  //otherwise returns 0x0 address
  function getTokenForEventId(uint16 eventId) public view returns (address tokenAddress);
  //returns the total count of all previous and current token statuses for the subscribed access address
  function getTotalStatusCounts(address tokenAddress) view public returns (uint16 errorsCount);
  //returns highest (current) token status level for the subscribed access address
  function getStatusLevel(address tokenAddress) view public returns (uint8 errorLevel);
  //returns current token status details for the subscribed access address
  function getCurrentStatusDetails(address tokenAddress) view public returns (
    uint8, // errorLevel,
    string, // errorMessage,
    address, // setter,
    uint); // timestamp);
  //returns token status details for the given statusNumber for the subscribed access address
  //statusMuber must be a number between 0 and return value of getTotalStatusCounts for the
  //given token
  function getStatusDetails(address tokenAddress, uint16 statusNumber) view public returns (
    uint8, // errorLevel,
    string, // errorMessage,
    address, // setter,
    uint, // timestamp,
    bool); // invalid);
  //returns last token status details for the subscribed access address
  //last StatusDetails do not have to be the current status details since
  //last status details can have lower error level than the previously set one
```

```
function getLastStatusDetails(address tokenAddress) view public returns (
    uint8, // errorLevel,
    string, // errorMessage,
    address, // setter,
    uint, // timestamp,
    bool); // invalid);

//checks if subscription is valid for the current subscriber address (address used for paying subscription)
function subscriptionIsValid() public view returns(bool isValid);
//checks if subscription is valid for the current access address
function subscriptionIsValidForAccessAddress() view public returns(bool isValid);
//checks if subscription is valid for the current subscriber address (address used for paying subscription)
function isExistingSubscriber() public view returns (bool isSubscriber);
//checks if current subscriber address (address used for paying subscription) is subscribed for the given token
function isSubscribedToToken(address token) public view returns (bool isSubscribed);
//checks if current access address is subscribed for the given token
function canAccessToken(address token) public view returns (bool canAccess);
//returns total number of supported token addresses
function getNumberSupportedTokens() public view returns (uint numberOfTokens);
//returns all supported token addresses
function getAllSupportedTokens() public view returns (address[] allTokens);
//returns number of remaining days for the subscription address
function remainingSubscriptionDays() public view returns (uint remainingDays);
//unsubscribes current subscription address, it does not refund paid subscription
function unsubscribe() public;
//calculates price to pay for the current subscriber address, takes into consideration current unused balance
//returns priceToPay, averageDailyPrice for the subscription and remainingOverheadBalance of the subscriber address
function calculatePrice(uint numberOfDays, uint numberTokens) view public returns (
    uint, // priceToPay,
    uint, // averageDailyPrice,
    uint); // remainingOverheadBalance);
//subscribes current address for Monitor chain for the array of passed tokens and number of days.
//number of days has to be hugher of equal to minDays. Subscribee address is accessAddress
//if the same address is used for subscription and for accessing MonitorChain the same address should
//be passed as subscribee argument. Payment should amount to the value returned by calculatePrice
function subscribe(address subscribee, uint numberOfDays, address[] tokenAddresses) public payable;
//subscribes current address for Monitor chain for all tokens and number of days.
//Number of days has to be hugher of equal to minDays. Subscribee address is accessAddress
//if the same address is used for subscription and for accessing MonitorChain the same address should
//be passed as subscribee argument
function subscribeAll(address subscribee, uint numberOfDays) public payable;
//returns subscription data for the subscriber address
function getSubscriptionData() public view returns (
    uint, // start,
    uint, // numberOfDays,
    uint, // dailyPrice,
    uint, // overheadBalance,
```

```
    address); // accessAddress);
  //returns if the passed address is blocked for the passed token for the subscribed access address
  function isAddressBlocked(address token, address addressToCheck) view public returns(bool);
  //token that is fired when status has been changed for a supported token
  //when such event is detected, getTokenForEventId function should be called
  //if the result is different from 0x0 address, it means that address is subscribed for the token
  //and its latest status can be retrieved using getLastStatusDetails for example
  event TokenStatusChanged(uint16 eventId);
}
```

Snippet 1 – AccessInterface function with detailed comments

Smart Contract Integration with MonitorChain Smart Contract

In order to support integration of a smart contract with MonitorChain smart contract, the address of a contract that is accessing MonitorChain has to be subscribed as access address. Accessing smart contract should be able to set MonitorChain contract address to its property in order to be able to invoke MonotorChain smart contract. In general, two main approaches for such integration exist, depending on the use case:

1. Smart Contract is directly accessing MonitorChain smart contract.
2. Proxy contract is accessing MonitorChain directly and at the same it is exposing functions that allows Accessing Smart Contact to react on token statuses.

Direct (Embedded) access to MonitorChain

Document SimpleTransfer.sol (found in /monitorchain-developer-sandbox-tools/ethereum/contracts/SimpleTransfer.sol) contains example of such integration. Accessing Smart Contract has to keep the address of the MonitorChain. Before checking the status of a specific token, accessing Smart Contract has to check if its subscription is still valid and if it is subscribed to a particular token. If it is, it can get the current token status, get the current token error or any status from the token history by calling appropriate functions of the MonitorChain Smart Contract that are exposed through the AccessInterface library.

```
contract SimpleTransfer {
  address public owner;
  address private monitorChain;
  function SimpleTransfer() public{
    owner = msg.sender;
  }
  modifier restrictToOwner(){
    require(msg.sender == owner);
    _;
  }
  function setMonitorChainAddress(address mcnAddress) public restrictToOwner {
    monitorChain = mcnAddress;
  }
  //anyone can trigger transfer from "from" to "to" as long as allowance has been set to the SimpleTransfer contract
  function transferFromToBlocking(address token, address from, address to, uint amount) public{
    require(ERC20Interface(token).allowance(from, address(this))>=amount);
    //check if for this address subscription is valid and if address is subscribed to the token it is checking
    if(monitorChain!=address(0) && AccessInterface(monitorChain).subscriptionIsValidForAccessAddress()
      && AccessInterface(monitorChain).canAccessToken(token))
      //check if MonitorChain returns an error for that token, different values can be used, like lower than 2
      require(AccessInterface(monitorChain).getStatusLevel(token) == 0);
    //if monitorChain does not return an error for the token or the current smart contract is not subscribed properly, execute transfer
    ERC20Interface(token).transferFrom(from,to,amount);
  }
```

Snippet 5 – Embedded MonitorChain integration based on token status

In the shown example (*Snippet 2*), if MonitorChain smart contract address is properly set, address of the contract is properly subscribed and can access the token in question, it can query MonitorChain for its status. The code can react depending on the retrieved status. In the current example, require command is used to block and revert the whole transaction effectively blocking the transfer on a particular token.

Similarly, depending on the need, executing function can be blocked only for the addresses that are blocked on the MonitorChain (*Snippet 3*) and not all the transactions on the token. The following example, after checking if the subscription is valid and if the contract address (access address) is subscribed for the token, checks then if the transfer source address is blocked on MonitorChain.

```
//anyone can trigger transfer from "from" to "to" as long as allowance has been set to the SimpleTransfer contract
//from address should be checked if it is being blocked
function transferFromToAddressBlocking(address token, address from, address to, uint amount) public{
    require(ERC20Interface(token).allowance(from, address(this))>=amount);
    //check if for this address subscription is valid and if address is subscribed to the token it is checking
    if(monitorChain!=address(0)
        && AccessInterface(monitorChain).subscriptionIsValidForAccessAddress()
        && AccessInterface(monitorChain).canAccessToken(token))
        //check if MonitorChain has blocked the address that is participating in the trasfer
        require(!AccessInterface(monitorChain).isAddressBlocked(token, from));
    //if monitorChain has not blocked the from address for the token or
    //the current smart contract is not subscribed properly, execute transfer
    ERC20Interface(token).transferFrom(from,to,amount);
}
```

Snippet 3 - Embedded MonitorChain integration based on blocked addresses for specific token

Intergration with MonitorChain through Proxy contract

Once dApp smart contract has been deployed to the Ethreum blockchain, it is impossible to change and therefore, each change requires new deployment and costly migration script. Therefore, it might be better option to integrate with MonitorChain through Proxy contract. Such proxy contract is an interface that exposes desired functions for the dApp smart contract. Actual Proxy interface implementation will use integration with MonitorChain to implement properly exposed functions. The example of a MonitorChain integration through the Proxy is given in the *SimpleTransferWithProxy.sol* document (found in /monitorchain-developer-sandbox-tools/ethereum/contracts/SimpleTransferWithProxy.sol).

In the example given in the following code snipped (*Snippet 4*), the *Proxy* contract exposes one function called *freeze*. The actual implementation of the interface is *MonitorChainProxy* smart contract. It implements the *freeze* function by checking the token status on the MonitorChain Smart Contract. The access address for accessing MonitorChain in this case must be the address of the *MonitorChainProxy* depoyed smart contract. Depending of the result to the MonitorChain *getStatusLevel* function, *freeze* function returns if transactions on the contract should be frozen or not.

```
contract Proxy{
   function freeze(address token) view public returns(bool);
}

contract MonitorChainProxy is Proxy{
   //user is the contract that is calling the proxy
   address private user;
   address private owner;
   address private monitorChain;
   function MonitorChainProxy(address userAddress) public{
      owner = msg.sender;
      user = userAddress;
   }
   function setMonitorChainAddress(address mcnAddress) public restrictToOwner {
      require(mcnAddress!=address(0));
      monitorChain = mcnAddress;
   }
   //important to add restriction modifier so MonitorChain data does not leak without subscription through the proxy to others
   function freeze(address token) view public restrictToUser returns(bool){
      if(monitorChain!=address(0)
         && AccessInterface(monitorChain).subscriptionIsValidForAccessAddress()
         && AccessInterface(monitorChain).canAccessToken(token))
            return AccessInterface(monitorChain).getStatusLevel(token) > 0;
      return false;
   }
```

Snippet 4 – Proxy for MonitorChain Smart Contract integration

After the *Proxy* and its exposed methods have been implemented, it should be properly connected with the dApp smart contract. Snippet 5 gives example how such Proxy contract can be used with dApp contract.

```
contract SimpleTransferWithProxy {
    address private owner;
    address private freezeProxy;
    function SimpleTransferWithProxy() public{
        owner = msg.sender;
    }
    modifier restrictToOwner(){
        require(msg.sender == owner);
        _;
    }
    function setFreezeProxy(address proxy) public restrictToOwner {
        freezeProxy = proxy;
    }
    //anyone can trigger transfer from "from" to "to" as long as allowance has been set to the SimpleTransfer contract
    function transferFromTo(address token, address from, address to, uint amount) public{
        require(ERC20Interface(token).allowance(from, address(this))>=amount);
        //check if for this address subscription is valid and if address is subscribed to the token it is checking
        if(freezeProxy!=address(0))
            //check if MonitorChain returns an error for that token, different values can be used, like lower than 2
            require(!Proxy(freezeProxy).freeze(token));
        //if monitorChain does not return an error for the token or the current smart contract is not
        //subscribed properly, execute the transaction
        ERC20Interface(token).transferFrom(from,to,amount);
    }
}
```

Snippet 5 – dApp Smart Contract using the Proxy exposed method

The Smart Contract must provide functions to set the Proxy contract address. Then, to check if the transaction should be executed, it calls *Proxy* contract exposed methods (*freeze* in this example) to check if a transaction on a token should be frozen. The *freeze* method itself, within the *Proxy* contract implementation calls MonitorChain Smart Contract to check the token status.

NodeJS script integration with MonitorChain

In order to enable off chain applications to react to token status changes (centralized exchanges, token tracking websites, news feeds), as well as to update other smart contracts states from off chain sources, integration with MonitorChain from off chain code has to be provided. One approach would be the use of the

AccessInterface.sol contract and proper wallet implementations and providers that can call the contract directly. Such approach is straightforward and is used in most cases when custom integration library does not exist. This instruction will not discuss such integration approach in particular. Using the proper compiled AccessInterface contract, wallet and web3 MonitorChain integration can be achieved.

In order to properly facilitate off chain MonitorChain, a custom NodeJS library that facilitates such integration is provided. The library can be retrieved from the github repo https://github.com/ZenchainSoftware/monitorchain-interface-library or from the https://www.npmjs.com/package/monitorchain-interface-library

The monitorchain-interface-library module can be easily installed using npm command:
$ npm install monitorchain-interface-library

Once installed, it easy to access MonitorChain using the AccessInterface wrapper. As given in the *Snippet 6*, the *AccessInterface* constructor accepts provider endpoint (for example http://rinkeby.infura.io/<API_KEY>), MonitorChain contract address and the mnemonic (seed) of the account that will be used to access the MonitorChain. Once the object has been created, it can call the functions of the AccessInterface in various ways, by resolving a promise, using a callback or just applying the async-await syntax.

```
const {AccessInterface} = require('monitorchain-interface-library');
const log = console.log;

const mc = new AccessInterface(
   'http://rinkeby.infura.io/<API KEY>',
   '0xF8CE9D2....71337Bd6201a', //The MonitorChain address
   '12 words mnemonic is here'
);

// Get the list of supported tokens (resolve a promise)
mc.getAllSupportedTokens().then(console.log);

// Get number of the supported tokens (using a callback)
mc.getNumberSupportedTokens((err, result) => {console.log(err, result)})
```

```
// Calculate a subscription price (async-await syntax)
const calc = async () => {
   // 45-days subscription for 50 tokens
   const price = await mc.calculatePrice(45, 50);
   console.log(price);
};

calc();
```

Snippet 6 – Examples of invoking AccessInterface javascript wrapper object functions

Similar example could be used for subscribing to the MonitorChain. The *AccessInterface* is using the built in truffle-hd-wallet which provides possibility to change the wallet index and therefore change the current address for the same set of mnemonics as provided on the Snippet 7.

```
const {AccessInterface} = require('monitorchain-interface-library');
const log = console.log;

const mc = new AccessInterface(
   'http://localhost:8545',
   '0xF8CE9D2...337Bd6201a',
   '12 words mnemonic is here'
);

const subscribe = async() => {
   mc.wallet = 2;

   log(await mc.getSubscriptionData());
   log(await mc.getTokensSubscribedTo());

   await mc.subscribe([
      "0xB8c77482e45F1F44dE1745F52C74426C631bDD52"
   ]);

   log(await mc.getSubscriptionData());
   log(await mc.remainingSubscriptionDays());
   log(await mc.getTokensSubscribedTo());
};

subscribe();
```

Snippet 7 – Subscribing the current wallet address to MonitorChain using AccessInterface library

Instead of initializing the *AccessInterface* by just passing arguments to its constructor, *AccessInterface* can be initialized with custom web3 object. In the following Snippet 8, the *AccessInterface* object is initialized with an existing web3 object and then used to query supported tokens on the MonitorChain.

```
const {AccessInterface} = require('monitorchain-interface-library');
const HDWalletProvider = require('truffle-hdwallet-provider');
const Web3 = require('web3');

const nodeAddress =  'http://localhost:8545';
const monitorChainAddress = '0xF8CE9D...7Bd6201a';
const mnemonic = '12 words mnemonic is here';

const web3 = new Web3(new HDWalletProvider(mnemonic, nodeAddress, 0, 20));

// A static method 'web3' allows to pass a custom web3 instance
const mc = AccessInterface.web3(web3, monitorChainAddress);

mc.getAllSupportedTokens(console.log);
```

Snippet 8 – Example of using custom web3 object

The most important part of MonitorChain integration off chain integration is listening and reacting to *TokenStatusChanged* events. Snippet 9 shows example of the code that listens to the token status changes.

```
const {AccessInterface, ERC20Interface} = require('monitorchain-interface-library');
const log = console.log;
const monitorChainAddress = '0xF8CE9D...37Bd6201a';

const mc = new AccessInterface(
   'http://localhost:8545',
   monitorChainAddress,
   '12 words mnemonic is here'
);

const ws = new AccessInterface(
   'ws://localhost:8543',
   monitorChainAddress
);

const callback = async (err, result) => {
   if(err) throw err;
   const tokenAddress = await mc.getTokenForEventId(result);
   if (!tokenAddress) return; // return if a customer is not subscribed to token

   log(`${tokenAddress}: a status has been changed: ${result}`);
   log(await mc.getCurrentStatusDetails(tokenAddress));
   const token =ERC20Interface.web3(mc.w3, tokenAddress);
   const tokenInfo = await token.tokenInfo();
   log(JSON.stringify(tokenInfo, null, 4));
};

ws.onStatusChanged(callback);
```

Snippet 9 – Example of listening to MonitorChain events

In order to allow for listening to events, Web Socket provider needs to be used when initializing the AccessInterface. The example uses two AccessInterface objects, one to access methods that check account subscription and the second one that is subscribed to public events on Ethereum blockchain. Once event is fired, its argument is the event Id. It is then used to retrieve the token which is associated with that event Id. If the current address, used to access MonitorChain, is not subscribed to the token whose status has been updated, the *getTokenForEventId* function call returns zero address. Otherwise it returns valid token address which then can be used to query for the latest token status.

Invoking token status changes

As mentioned earlier, integration sandbox is initialized with four token contract addresses. Token smart contracts were implemented in such a way to contain most known bugs (like bufferoverflow exploit) and allow sending from address to any other one. The breakable token contract can be found in the document *McnToken.sol* (the path is monitorchain-developer-sandbox-tools/ethereum/contracts/McnToken.sol).

However direct interaction with the token smart contract functions is not necessary. The sandbox version of the MonitorChain, instead of opening token smart contract code on Etherscan, (*Figure 5*) opens a page called token breaker that allows user to break the token. Such transfer will be detected by the monitor and appropriate status will be updated on the MonitorChain.

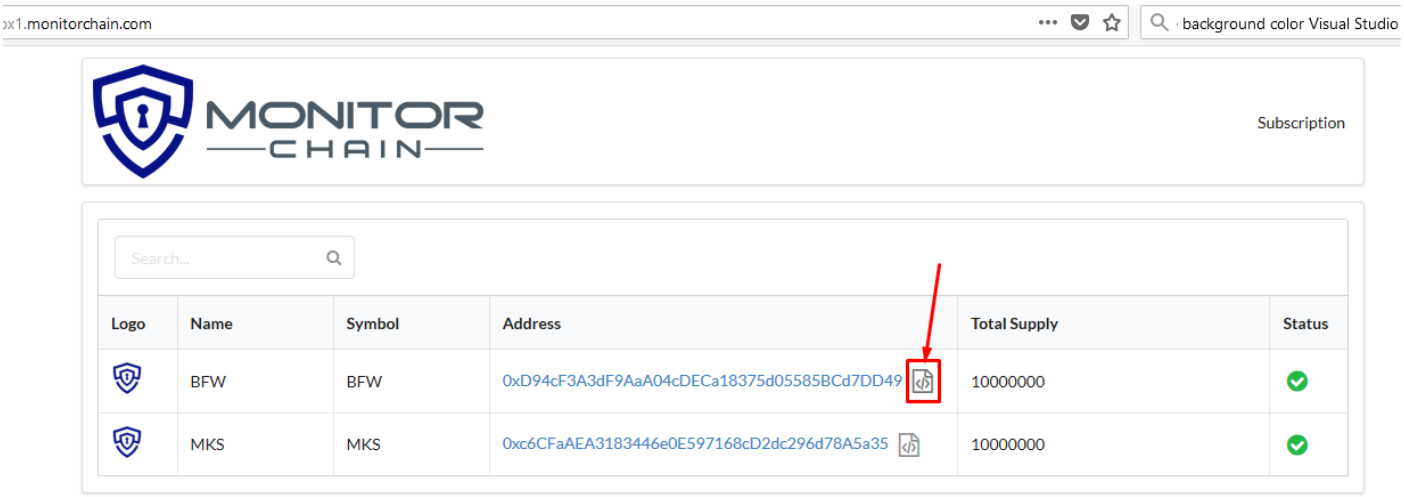

Figure 5 – Opening token breaker page of the sandbox MonitorChain version

Token breaker page allows user to execute various token exploits that were modeled by various historical token thefts like bufferoverflow, excessive token minting etc. Each functionality on the token breaker should be supplied with the transfer destination address (*Figure 6*).

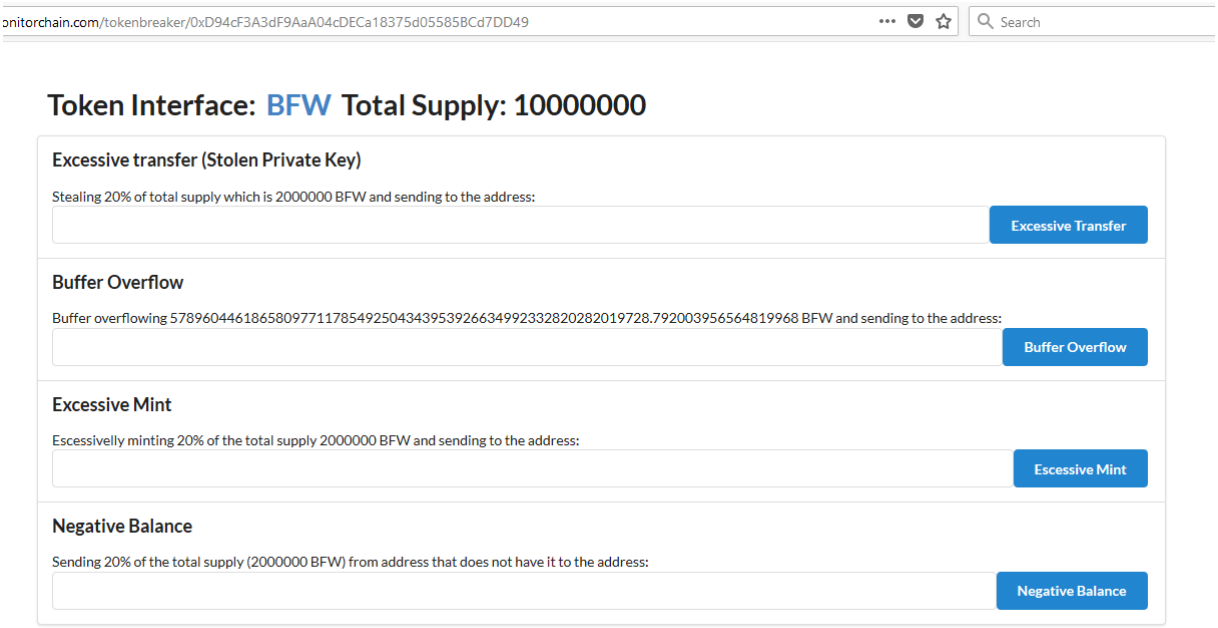

Figure 6 – Token breaker page

In cases when it is not required that the token status change comes from the monitor itself due to a detected malicious transaction on a token, user is able to set token error via admin panel. Sandbox package is supplied with a set of mnemonic for which the account with index 0 is the sandbox admin. Admin can access admin panel and through it change the token status (set new status or clear the status). When admin navigates to the sandbox MonitorChain homepage (for example sandbox1.monitorchain.com), she will have admin menu link in the top right corner of the app (*Figure 7*).

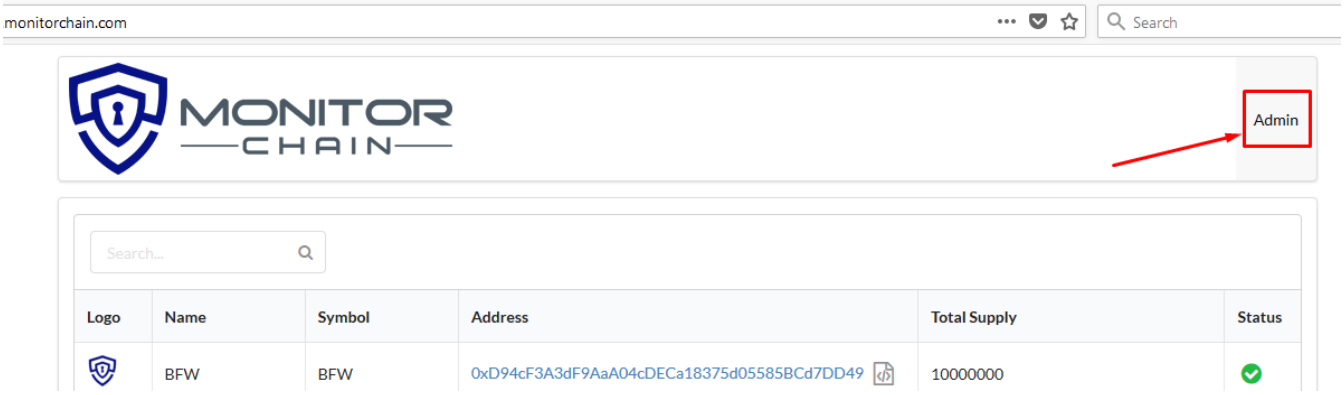

Figure 7 – Admin menu item

After clicking on the admin menu item, user is navigated to the admin panel where she can check existing subscribers and supported tokens. On the tokens tab, admin can review the tokens, add support for new ones or remove exiting ones. After clicking on desired token address, it details load in the token panel which allows admin to set or clear the error status of the token (*Figure 8*). Such status change also triggers *TokenStatusChanged* event on the MonitorChain Smart Contract which can then be detected using the *AccessInterface* library.

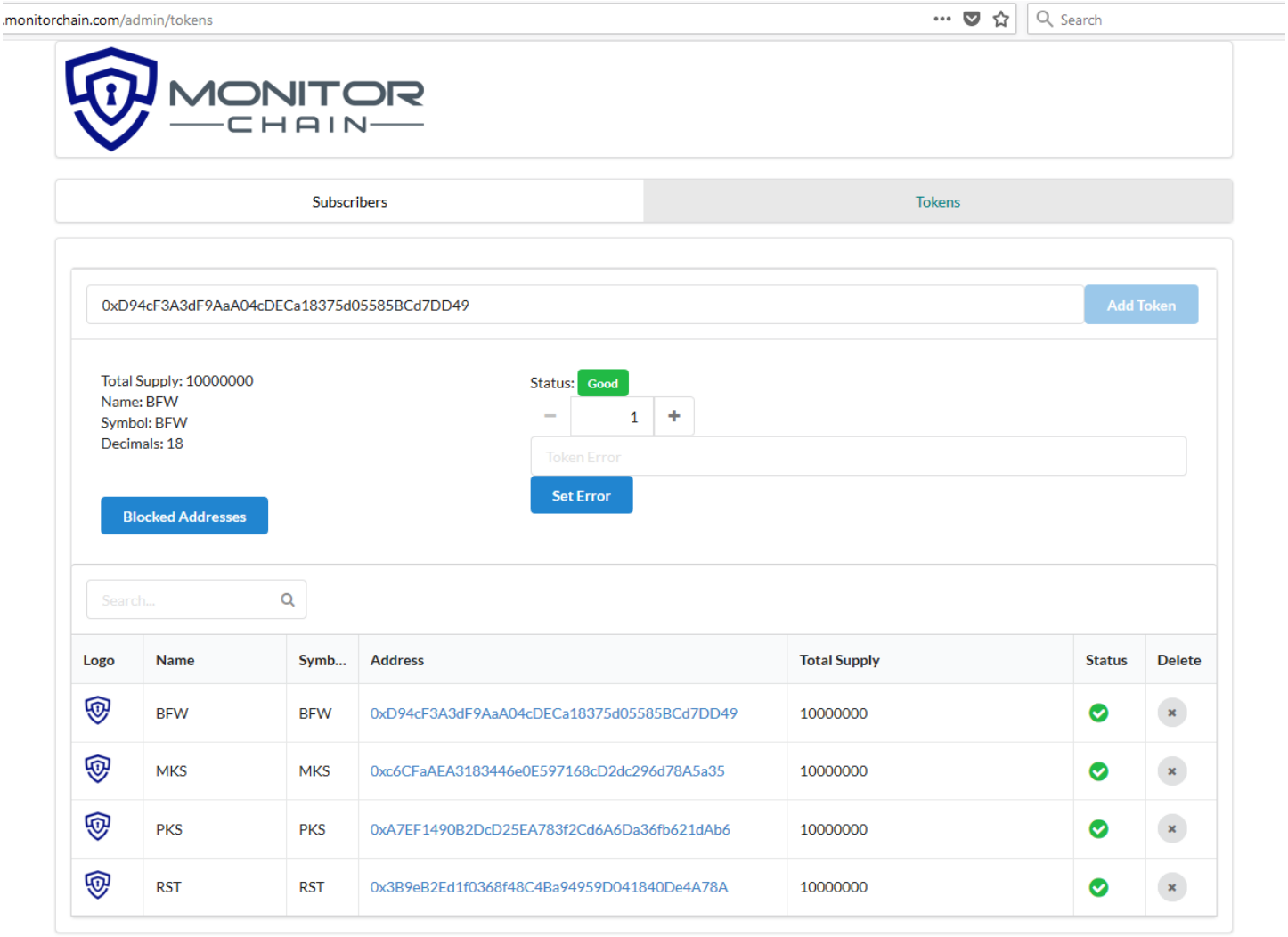

Figure 7 – Token admin panel

Resetting sandbox MonitorChain

After working for some time with MonitorChain sandbox version, user might want to reset it to start over. In order to support that feature, MonitorChain reset script is provided on the github https://github.com/ZenchainSoftware/monitorchain-developer-sandbox-tools. All user has to do is to call reset.js script with appropriate admin mnemonics. The scrip will reset the token statuses on the MonitorChain.

Additional resources

Additional resources that can be used to integration can be found on the official github account of the ZenChain Inc. https://github.com/ZenchainSoftware . For all additional question Zenchain developers are at your disposal. Please don't hesitate to reach out and schedule consolations with our development team.