FXY TOKEN SMART CONTRACT AUDIT RESULTS

FOR FIXY NETWORK LTD

04/05/2018

Made in Germany by chainsulting.de

Change history

Version	Date	Author	Changes
1.0	05.04.2017	Chainsulting	Audit created

Smart Contract Audit Fixy Network ICO

Table of Contents

1. Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information, presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of Fixy Network Ltd. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

2. Vulnerability Level

1-Low severity - A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.

2-Medium severity – A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.

3-High severity – A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.

4-Critical severity – A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.

3. Overview of the audit

The project has two files, the FixyNetwork.sol file which contains 295 lines of Solidity code and crowdsale.sol file which contains 208 lines of Solidity code. All the functions and state variables are well commented using the natspec documentation for the functions which is good to understand quickly how everything is supposed to work.

You can find the files on github:
https://github.com/fixynetwork/FIXY-NETWORK/tree/master/Solidity

Etherscan:

FXY Token address:
https://etherscan.io/token/0xa024e8057eec474a9b2356833707dd0579e26ef3

Token Code address:
https://etherscan.io/address/0xa024e8057eec474a9b2356833707dd0579e26ef3#code

Token Crowdsale address:
https://etherscan.io/address/0xac36d88dffc8d596c3b5a496a76cf5d274100a85#code

Token Holder and Creator address:
https://etherscan.io/address/0xe65b6eeafe34adb2e19e8b2ae9c517688771548e

Project Website:
https://www.fixyapp.io

Token Standard
ERC20

Token burning mechanics
Supported

Solidity Version:
FixyNetwork.sol 4.0.12
Crowdsale.sol 4.0.11

Token minting mechanics
Not supported

Min/max contribution
Not supported

Token bonus structure
PRE-ICO. (1ETH= 7000 FXY)
Start 1.5.2018
End 9.5.2018
Total coins with 40% bonus 14.000.000 FXY

ICO LEVEL 1 (1ETH=6000 FXY)
Start 16.5.2018
End 23.5.2018
Total coins 54.000.000 FXY with 20% bonus

ICO LEVEL 2 (1ETH=5000FXY)
Start 25.5.2018
End 31.5.2018

Used Code from other Smart Contracts

1. SafeMath (Math operations with safety checks that throw on error)
https://github.com/OpenZeppelin/zeppelin-solidity/tree/master/contracts/math

2. ERC20Basic (Simpler version of ERC20 interface)
https://github.com/ethereum/EIPs/issues/179
https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/token/ERC20/ERC20Basic.sol

3. Standard ERC20 (Based on code by FirstBlood)
https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol

4. Burnable Token (Token that can be irreversibly burned (destroyed)
https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/token/ERC20/BurnableToken.sol

5. Crowdsale (Crowdsale is a base contract for managing a token crowdsale. Crowdsales have a start and end timestamps, where investors can make token purchases and the crowdsale will assign them tokens based on a token per ETH rate. Funds collected are forwarded to a wallet as they arrive.)
https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/crowdsale/Crowdsale.sol
Crowdsale.sol

4. Attack made to the contract

Security Report

Attack: Transactions May Affect Ether Receiver
A contract is exposed to this vulnerability if a miner (who executes and validates transactions) can reorder the transactions within a block in a way that affects the receiver of ether.

Severity: 1

Result / Recommendation: Both contracts are not vulnerable to this vulnerability as the receiver of ether is **msg.sender**, which cannot be modified by previously executed transactions.

Attack: Hardcoded address
The contract contains unknown address. This address might be used for some malicious activity.

Solidity/crowdsale.sol linie 83
wallet = 0xe65b6eEAfE34adb2e19e8b2AE9c517688771548E;

Solidity/crowdsale.sol linie 86
addressOfTokenUsedAsReward =
0xA024E8057EEC474a9b2356833707Dd0579E26eF3;

Severity: 1

Result / Recommendation:
Both addresses are owned and verified by Fixy Network Limited

---

Attack: Using the approve function of the ERC-20 standard
The approve function of ERC-20 might lead to vulnerabilities.

Solidity/FixyNetwork.sol linie 210 - 214

```
function approve(address _spender, uint256 _value) public returns (bool) {
    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
    return true;
  }
```

Severity: 3

Result / Recommendation:
Only use the approve function of the ERC-20 standard to change allowed amount to 0 or from 0 (wait till transaction is mined and approved).

Attack: Costly loop
Ethereum is a very resource-constrained environment. Prices per computational step are orders of magnitude higher than with centralized providers. Moreover, Ethereum miners impose a limit on the total number of gas consumed in a block. If array.length is large enough, the function exceeds the block gas limit, and transactions calling it will never be confirmed:

```
for (uint256 i = 0; i < array.length ; i++) {
        cosltyFunc();
}
```

This becomes a security issue, if an external actor influences array.length. E.g., if array enumerates all registered addresses, an adversary can register many addresses, causing the problem described above.

Solidity/crowdsale.sol linie 128-129 / 122 -123

```
for(uint i = 0;i < _addrs.length;++i)
    whitelist[_addrs[i]] = false;
```

Severity: 2

Result / Recommendation:
Avoid loops with big or unknown number of steps.

Attack: Compiler version not fixed
Solidity source files indicate the versions of the compiler they can be compiled with.

```
pragma solidity ^0.4.17; // bad: compiles w 0.4.17 and above
pragma solidity 0.4.17; // good : compiles w 0.4.17 only
```

It is recommended to follow the latter example, as future compiler versions may handle certain language constructions in a way the developer did not foresee.

Solidity/FixyNetwork.sol linie 1

```
pragma solidity ^0.4.12;
```

Solidity/crowdsale.sol linie 1

```
pragma solidity ^0.4.11;
```

Severity: 1

Result / Recommendation:
Specify the exact compiler version (pragma solidity x.y.z;).

You’re specifying a pragma version with the caret symbol (^) up front which tells the compiler to use any version of solidity bigger than 0.4.11 .

This is not a good practice since there could be major changes between versions that would make your code unstable. That’s why I recommend to set a fixed version without the caret like 0.4.11.

Attack: Timestamp dependence
The timestamp of the block can be slightly manipulated by the miner. One should not use timestamp's exact value for critical components of the contract.

Solidity/Crowdsale.sol linie 79

startTime = now + 80715 * 1 minutes;

Severity: 3

Result / Recommendation:
Block numbers and average block time can be used to estimate time, but this is not future proof as block times may change (such as the changes expected during Casper). Do not use now for randomness.

Attack: Unchecked math
Solidity is prone to integer over- and underflow. Overflow leads to unexpected effects and can lead to loss of funds if exploited by a malicious account.

File: Solidity/FixyNetwork.sol
Lines: 285-285
File: Solidity/crowdsale.sol
Lines: 79-79
Lines: 80-80
Lines: 149-149
Lines: 168-168
Lines: 169-169
Lines: 170-170
Lines: 171-171
Lines: 173-173
Lines: 174-174
Lines: 175-175

Severity: 3

Result / Recommendation:
Check against over- and underflow (use the SafeMath library).
SafeMath is already used in both contracts.

5. Executive Summary

A majority of the code was standard and copied from widely-used and reviewed contracts and, as a result, a lot of the code was reviewed before. It correctly implemented widely-used and reviewed contracts for safe mathematical operations. The audit identified no security vulnerabilities.

6. General Summary

The issues identified were minor in nature, and do not affect the security of the contract. The code specifies Solidity version 0.4.11 and .12 , which has only recently had a newer version of 0.4.19 released. As a result, Fixy Network should consider updating the pragma statements to require the latest version of Solidity.

Additionally, the code implements and uses a SafeMath contract, which defines functions for safe math operations that will throw errors in the cases of integer overflow or underflows. The simplicity of the audited contracts contributed greatly to their security. The minimalist approach in choosing which pieces of functionality to implement meant there was very little attack surface available.

Solidity Version Updates Recommended

Most of the code uses Solidity 0.4.11 and 0.4.12, but some files under Ownership are marked 0.4.0. These should be updated.

Solidity 0.4.10 will add several features which could be useful in these contracts:

- assert(condition), which throws if the condition is false
- revert(), which rolls back without consuming all remaining gas.
- address.transfer(value), which is like send but automatically propagates exceptions, and supports .gas(). See https://github.com/ethereum/solidity/issues/610 for more on this

7. Source Code – Smart Contracts

FixyNetwork.sol

```
pragma solidity ^0.4.12;

/**
* @title SafeMath
* @dev Math operations with safety checks that throw on error
*/
library SafeMath {
function mul(uint256 a, uint256 b) internal constant returns (uint256) {
uint256 c = a * b;
assert(a == 0 || c / a == b);
return c;
}

function div(uint256 a, uint256 b) internal constant returns (uint256) {
// assert(b > 0); // Solidity automatically throws when dividing by 0
uint256 c = a / b;
// assert(a == b * c + a % b); // There is no case in which this doesn't hold
return c;
}

function sub(uint256 a, uint256 b) internal constant returns (uint256) {
assert(b <= a);
return a - b;
}

function add(uint256 a, uint256 b) internal constant returns (uint256) {
uint256 c = a + b;
assert(c >= a);
return c;
}
}

/**
```

```
* @title Ownable
* @dev The Ownable contract has an owner address, and provides basic
authorization control
* functions, this simplifies the implementation of "user permissions".
*/
contract Ownable {
address public owner;

event OwnershipTransferred(address indexed previousOwner, address indexed
newOwner);

/**
* @dev The Ownable constructor sets the original `owner` of the contract to the
sender
* account.
*/
function Ownable() {
owner = msg.sender;
}

/**
* @dev Throws if called by any account other than the owner.
*/
modifier onlyOwner() {
require(msg.sender == owner);
_;
}

/**
* @dev Allows the current owner to transfer control of the contract to a newOwner.
* @param newOwner The address to transfer ownership to.
*/
function transferOwnership(address newOwner) onlyOwner public {
require(newOwner != address(0));
OwnershipTransferred(owner, newOwner);
```

```
owner = newOwner;
}
}

/**
* @title ERC20Basic
* @dev Simpler version of ERC20 interface
* @dev see https://github.com/ethereum/EIPs/issues/179
*/
contract ERC20Basic {
uint256 public totalSupply;
function balanceOf(address who) public constant returns (uint256);
function transfer(address to, uint256 value) public returns (bool);
event Transfer(address indexed from, address indexed to, uint256 value);
}

/**
* @title Basic token
* @dev Basic version of StandardToken, with no allowances.
*/
contract BasicToken is ERC20Basic, Ownable {
using SafeMath for uint256;

mapping(address => uint256) balances;
// allowedAddresses will be able to transfer even when locked
// lockedAddresses will *not* be able to transfer even when *not locked*
mapping(address => bool) public allowedAddresses;
mapping(address => bool) public lockedAddresses;
bool public locked = true;

function allowAddress(address _addr, bool _allowed) public onlyOwner {
require(_addr != owner);
allowedAddresses[_addr] = _allowed;
}

function lockAddress(address _addr, bool _locked) public onlyOwner {
```

```
require(_addr != owner);
lockedAddresses[_addr] = _locked;
}

function setLocked(bool _locked) public onlyOwner {
locked = _locked;
}

function canTransfer(address _addr) public constant returns (bool) {
if(locked){
if(!allowedAddresses[_addr]&&_addr!=owner) return false;
}else if(lockedAddresses[_addr]) return false;

return true;
}


/**
* @dev transfer token for a specified address
* @param _to The address to transfer to.
* @param _value The amount to be transferred.
*/
function transfer(address _to, uint256 _value) public returns (bool) {
require(_to != address(0));
require(canTransfer(msg.sender));

// SafeMath.sub will throw if there is not enough balance.
balances[msg.sender] = balances[msg.sender].sub(_value);
balances[_to] = balances[_to].add(_value);
Transfer(msg.sender, _to, _value);
return true;
}

/**
* @dev Gets the balance of the specified address.
* @param _owner The address to query the the balance of.
* @return An uint256 representing the amount owned by the passed address.
*/
function balanceOf(address _owner) public constant returns (uint256 balance) {
return balances[_owner];
}
}

/**
* @title ERC20 interface
* @dev see https://github.com/ethereum/EIPs/issues/20
*/
contract ERC20 is ERC20Basic {
function allowance(address owner, address spender) public constant returns
(uint256);
function transferFrom(address from, address to, uint256 value) public returns
(bool);
function approve(address spender, uint256 value) public returns (bool);
event Approval(address indexed owner, address indexed spender, uint256 value);
}

/**
* @title Standard ERC20 token
*
* @dev Implementation of the basic standard token.
* @dev https://github.com/ethereum/EIPs/issues/20
* @dev Based on code by FirstBlood:
https://github.com/Firstbloodio/token/blob/master/smart_contract/FirstBloodToken.
sol
*/
contract StandardToken is ERC20, BasicToken {

mapping (address => mapping (address => uint256)) allowed;

/**
* @dev Transfer tokens from one address to another
* @param _from address The address which you want to send tokens from
* @param _to address The address which you want to transfer to
* @param _value uint256 the amount of tokens to be transferred
*/
function transferFrom(address _from, address _to, uint256 _value) public returns
(bool) {
require(_to != address(0));
require(canTransfer(msg.sender));

uint256 _allowance = allowed[_from][msg.sender];

// Check is not needed because sub(_allowance, _value) will already throw if this
condition is not met
// require (_value <= _allowance);

balances[_from] = balances[_from].sub(_value);
balances[_to] = balances[_to].add(_value);
allowed[_from][msg.sender] = _allowance.sub(_value);
Transfer(_from, _to, _value);
return true;
}

/**
* @dev Approve the passed address to spend the specified amount of tokens on
behalf of msg.sender.
*
* Beware that changing an allowance with this method brings the risk that
someone may use both the old
* and the new allowance by unfortunate transaction ordering. One possible
solution to mitigate this
* race condition is to first reduce the spender's allowance to 0 and set the desired
value afterwards:
* https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint256 _value) public returns (bool) {
allowed[msg.sender][_spender] = _value;
Approval(msg.sender, _spender, _value);
```

```
return true;

}

/**

* @dev Function to check the amount of tokens that an owner allowed to a
spender.

* @param _owner address The address which owns the funds.

* @param _spender address The address which will spend the funds.

* @return A uint256 specifying the amount of tokens still available for the spender.

*/

function allowance(address _owner, address _spender) public constant returns
(uint256 remaining) {

return allowed[_owner][_spender];

}

/**

* approve should be called when allowed[_spender] == 0. To increment

* allowed value is better to use this function to avoid 2 calls (and wait until

* the first transaction is mined)

* From MonolithDAO Token.sol

*/

function increaseApproval (address _spender, uint _addedValue)

returns (bool success) {

allowed[msg.sender][_spender] =
allowed[msg.sender][_spender].add(_addedValue);

Approval(msg.sender, _spender, allowed[msg.sender][_spender]);

return true;

}

function decreaseApproval (address _spender, uint _subtractedValue)

returns (bool success) {

uint oldValue = allowed[msg.sender][_spender];

if (_subtractedValue > oldValue) {

allowed[msg.sender][_spender] = 0;

} else {

allowed[msg.sender][_spender] = oldValue.sub(_subtractedValue);
}
Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
return true;
}
}

/**
* @title Burnable Token
* @dev Token that can be irreversibly burned (destroyed).
*/
contract BurnableToken is StandardToken {

event Burn(address indexed burner, uint256 value);

/**
* @dev Burns a specific amount of tokens.
* @param _value The amount of token to be burned.
*/
function burn(uint256 _value) public {
require(_value > 0);
require(_value <= balances[msg.sender]);
// no need to require value <= totalSupply, since that would imply the
// sender's balance is greater than the totalSupply, which *should* be an assertion
failure

address burner = msg.sender;
balances[burner] = balances[burner].sub(_value);
totalSupply = totalSupply.sub(_value);
Burn(burner, _value);
Transfer(burner, address(0), _value);
}
}

contract FixyNetwork is BurnableToken {

string public constant name = "FIXY NETWORK";
string public constant symbol = "FXY";
```

```
uint public constant decimals = 18;
// there is no problem in using * here instead of .mul()
uint256 public constant initialSupply = 100000000 * (10 ** uint256(decimals));

// Constructors
function FixyNetwork () {
totalSupply = initialSupply;
balances[msg.sender] = initialSupply; // Send all tokens to owner
allowedAddresses[owner] = true;
}


}
```

crowdsale.sol

```
pragma solidity ^0.4.11;

/**
* @title SafeMath
* @dev Math operations with safety checks that throw on error
*/
library SafeMath {
function mul(uint256 a, uint256 b) internal constant returns (uint256) {
uint256 c = a * b;
assert(a == 0 || c / a == b);
return c;
}

function div(uint256 a, uint256 b) internal constant returns (uint256) {
// assert(b > 0); // Solidity automatically throws when dividing by 0 uint256 c = a /
b;
uint256 c = a / b;
// assert(a == b * c + a % b); // There is no case in which this doesn't hold
return c;
}
```

```
function sub(uint256 a, uint256 b) internal constant returns (uint256) {
assert(b <= a);
return a - b;
}

function add(uint256 a, uint256 b) internal constant returns (uint256) {
uint256 c = a + b;
assert(c >= a);
return c;
}
}

/**
* @title Crowdsale
* @dev Crowdsale is a base contract for managing a token crowdsale.
* Crowdsales have a start and end timestamps, where investors can make
* token purchases and the crowdsale will assign them tokens based
* on a token per ETH rate. Funds collected are forwarded
to a wallet
* as they arrive.
*/
contract token { function transfer(address receiver, uint amount){ } }
contract Crowdsale {
using SafeMath for uint256;

// uint256 durationInMinutes;
// address where funds are collected
address public wallet;
// token address
address public addressOfTokenUsedAsReward;

// uint256 public price = 18000;

token tokenReward;

// mapping (address => uint) public contributions;
mapping(address => bool) public whitelist;
```

```
// start and end timestamps where investments are allowed (both inclusive)
uint256 public startTime;
uint256 public endTime;
// amount of raised money in wei
uint256 public weiRaised;
uint256 public tokensSold;

/**
* event for token purchase logging
* @param purchaser who paid for the tokens
* @param beneficiary who got the tokens
* @param value weis paid for purchase
* @param amount amount of tokens purchased
*/
event TokenPurchase(address indexed purchaser, address indexed beneficiary,
uint256 value, uint256 amount);


function Crowdsale() {
// how many minutes
startTime = now + 80715 * 1 minutes;
endTime = startTime + 31*24*60*1 minutes;

//You will change this to your wallet where you need the ETH
wallet = 0xe65b6eEAfE34adb2e19e8b2AE9c517688771548E;
// durationInMinutes = _durationInMinutes;
//Here will come the checksum address we got
addressOfTokenUsedAsReward =
0xA024E8057EEC474a9b2356833707Dd0579E26eF3;


tokenReward = token(addressOfTokenUsedAsReward);
}

// bool public started = true;

// function startSale(){
// require(msg.sender == wallet);
// started = true;
// }

// function stopSale(){
// require(msg.sender == wallet);
// started = false;
// }

// function setPrice(uint256 _price){
// require(msg.sender == wallet);
// price = _price;
// }

function changeWallet(address _wallet){
require(msg.sender == wallet);
wallet = _wallet;
}

// function changeTokenReward(address _token){
// require(msg.sender==wallet);
// tokenReward = token(_token);
// addressOfTokenUsedAsReward = _token;
// }

function whitelistAddresses(address[] _addrs){
require(msg.sender==wallet);
for(uint i = 0; i < _addrs.length; ++i)
whitelist[_addrs[i]] = true;
}

function removeAddressesFromWhitelist(address[] _addrs){
require(msg.sender==wallet);
for(uint i = 0;i < _addrs.length;++i)
whitelist[_addrs[i]] = false;
}

// fallback function can be used to buy tokens
```

```
function () payable {
buyTokens(msg.sender);
}

// low level token purchase function
function buyTokens(address beneficiary) payable {
require(beneficiary != 0x0);
require(validPurchase());
require(whitelist[beneficiary]);

uint256 weiAmount = msg.value;

// if(weiAmount < 10**16) throw;
// if(weiAmount > 50*10**18) throw;

// calculate token amount to be sent
uint256 tokens = (weiAmount) * 5000;//weiamount * price
// uint256 tokens = (weiAmount/10**(18-decimals)) * price;//weiamount * price

//bonus schedule

/*
PRE-ICO. (1ETH= 7000 FXY)
Start 1.5.2018
End 9.5.2018
Total coins with 40% bonus 14.000.000 FXY
ICO LEVEL 1 (1ETH=6000 FXY)
Start 16.5.2018
End 23.5.2018
Total coins 54.000.000 FXY with 20% bonus
ICO LEVEL 2 (1ETH=5000FXY)
Start 25.5.2018
End 31.5.2018
Total coins —> if on ICO Level 1 not sold out, it will be drop here.
*/
if(now < startTime + 9*24*60* 1 minutes){
tokens += (tokens * 40) / 100;//40%
if(tokensSold>14000000*10**18) throw;
}else if(now < startTime + 16*24*60* 1 minutes){
throw;
}else if(now < startTime + 23*24*60* 1 minutes){
tokens += (tokens * 20) / 100;
}else if(now < startTime + 25*24*60* 1 minutes){
throw;
}

// update state
weiRaised = weiRaised.add(weiAmount);

// if(contributions[msg.sender].add(weiAmount)>10*10**18) throw;
// contributions[msg.sender] = contributions[msg.sender].add(weiAmount);

tokenReward.transfer(beneficiary, tokens);
tokensSold = tokensSold.add(tokens);
TokenPurchase(msg.sender, beneficiary, weiAmount, tokens);
forwardFunds();
}

// send ether to the fund collection wallet
// override to create custom fund forwarding mechanisms
function forwardFunds() internal {
wallet.transfer(msg.value);
}

// @return true if the transaction can buy tokens
function validPurchase() internal constant returns (bool) {
bool withinPeriod = now >= startTime && now <= endTime;
bool nonZeroPurchase = msg.value != 0;
return withinPeriod && nonZeroPurchase;
}

function withdrawTokens(uint256 _amount) {
require(msg.sender==wallet);
tokenReward.transfer(wallet,_amount);
```

```
 }
}
```

ERC20 API: An Attack Vector on Approve/TransferFrom Methods

0. Abstract

In this article we describe a possible attack vector on standard ERC20 Ethereum Token API. This is attack on API itself, not on any particular implementations, so all conformant implementations are potentially vulnerable. The method uses methods approve and transferFrom defined by ERC20. We also give some thoughts about how described attack could be prevented or at least mitigated using current version of ERC20 API. We also suggest changes to ERC20 API that would make the described attack impossible.

1. Introduction

ERC20, defines a standard API for Ethereum Tokens smart contracts. Tokens are defined by Ethereum Foundation as the following:

Tokens in the ethereum ecosystem can represent any fungible tradable good: coins, loyalty points, gold certificates, IOUs, in game items, etc. Since all tokens implement some basic features in a standard way, this also means that your token will be instantly compatible with the ethereum wallet and any other client or contract that uses the same standards.

So ERC20 is supposed to be the standard way to implement basic features of all tokens to make them compatible with common Ethereum software such as Ethereum Wallet.

2. Approve/TransferFrom Methods

Among other things, ERC20 defines the following two methods to be implemented by every Ethereum Token smart contract:

```
function transferFrom(address _from, address _to, uint256 _value)
returns (bool success)
```

Send `_value` amount of tokens from address `_from` to address `_to`
The `transferFrom` method is used for a withdraw workflow, allowing contracts to send tokens on your behalf, for example to "deposit" to a contract address and/or to charge fees in sub-currencies; the command should fail unless the `_from` account has deliberately authorized the sender of the message via some mechanism; we propose these standardized APIs for approval:

```
function approve(address _spender, uint256 _value)
returns (bool success)
```

Allow `_spender` to withdraw from your account, multiple times, up to the `_value` amount. If this function is called again it overwrites the current allowance with `_value`.

Apart from updating the allowance, the ERC-20 proposal does not specify the intended semantics of multiple calls to `approve`.

In the following sections we will show how these two methods, as defined in ERC20, could be used in an attack that allows a spender to transfer more tokens than the owner of the tokens ever wanted to allow the spender to transfer.

3. Attack Scenario

Here is possible attack scenario:

1. Alice allows Bob to transfer N of Alice's tokens (N>0) by calling `approve` method on Token smart contract passing Bob's address and N as method arguments
2. After some time, Alice decides to change from N to M (M>0) the number of Alice's tokens Bob is allowed to transfer, so she calls `approve` method again, this time passing Bob's address and M as method arguments
3. Bob notices Alice's second transaction before it was mined and quickly sends another transaction that calls `transferFrom` method to transfer N Alice's tokens somewhere
4. If Bob's transaction will be executed before Alice's transaction, then Bob will successfully transfer N Alice's tokens and will gain an ability to transfer another M tokens
5. Before Alice noticed that something went wrong, Bob calls `transferFrom` method again, this time to transfer M Alice's tokens.

So, Alice's attempt to change Bob's allowance from N to M (N>0 and M>0) made it possible for Bob to transfer N+M of Alice's tokens, while Alice never wanted to allow so many of her tokens to be transferred by Bob.

4. Attack Analysis

The attack described above is possible because `approve` method overrides current allowance regardless of whether spender already used it or not, so there is no way to increase or decrease allowance by certain value atomically, unless token owner is a smart contract, not an account.

5. Workaround

Because the described attack allows an attacker to transfer at most N + M tokens when allowance is being changed from N to M, then changing allowance from N to 0 and then from 0 to M seems quite safe. Token owner just needs to make sure that first transaction actually changed allowance from N to 0, i.e. that the spender didn't manage to transfer some of N allowed tokens before first transaction was mined. Unfortunately, such checking does not seem to be possible via standard Web3 API, because to do the check one needs to be able to analyze changes in the storage of smart contract made by particular transactions,

including internal transactions. Though, such checking is still possible using advanced blockchain explorers such as EtherCamp.
Another way to mitigate the threat is to approve token transfers only to smart contracts with verified source code that does not contain logic for performing attacks like described above, and to accounts owned by the people you may trust.

6. Suggested ERC20 API Changes

This section suggests changes to ERC20 API that are supposed to make the attack described above impossible.

6.1. Atomic "Compare And Set" Approve Method

We suggest the following method to be added to ERC20 API:

```
function approve(
  address _spender,
  uint256 _currentValue,
  uint256 _value)
returns (bool success)
```

If current allowance for `_spender` is equal to `_currentValue`, then overwrite it with `_value` and return `true`, otherwise return `false`.

This change alone is enough to address the attack vector described above. Suggestions given below are not required, but are supposed to make usage of `approve` and `transferFrom` methods more convenient and less error-prone.

6.2. Separate Log Message for "TransferFrom" Transfers

We suggest the following event to be added to ERC20 API:

```
event Transfer(
  address indexed _spender,
  address indexed _from,
  address indexed _to,
  uint256 _value)
```

Triggered when tokens are transferred via `transferFrom` method.

Note, that for backward compatibility reasons, token contracts will probably have to log both, old three-args and new four-args `Transfer` events in `transferFrom` method.

6.3. Four-Args Approval Event

We suggest the following event to be added to ERC20 API:

```
event Approval(
  address indexed _owner,
  address indexed _spender,
  uint256 _oldValue,
  uint256 _value)
```

Triggered whenever either `approve` method is called.

Note, that for backward compatibility reasons, token contract will probably have to log both, old three-args and new four-args `Approval` events in `approve` method.