Audit of the smart-contracts (BitWork)

Status – in work.

Code of the contracts is checked for critical errors which could lead to losses of money of the investors.
Contracts use methods of the library SafeMath for safe calculations.

Summary:

Critical errors:
Errors that can lead to loss of funds, as well as problems that violate the main functionality of contracts.
Found: 0.

Medium errors:
Errors of logic and functionality of contracts that do not lead to loss of funds.
Found: 6.

Optimization:
Opportunities to reduce transaction costs and reduce the number of lines of code.
Found: 8.

Notes and recommendations.:
Tips and tricks, as well as errors that do not affect the functionality of the smart contract.
Found: 9.

Contract CourseContract

The main purpose of the functionality of the **CourseContract** is selling educational courses for cryptocurrencies (ETH).

Contract inherits from contracts **Adminable** and **Ownable**.

The functionality of the contract **Adminable** implements the access restriction to functions of the contract by the modifier **onlyAdminAndOwner**.

The modifier **onlyAdminAndOwner** restricts the calling functions with roles owner or admin. The admin role can be set via the **setAdmin** function, which is available only to the owner role (via the **onlyOwner** modifier implemented in the **Ownable** contract).

The functionality of the contract **Ownable** implements the access restriction functions of the contract by the modifier **onlyOwner**.

Modifier **onlyOwner** restricts the function call with the owner role. The owner role is set when the contract is deployed (with the value of the address which deployed it). The new owner value can be set by calling the **transferOwnership** function through the address **newOwner** parameter. The **newOwner** value cannot be zero.

Variables:

1. name:
   name;
   set to "BitWork Course Sale Conrtact" by default;

   - Note: possible typo in the word “Contract”;

2. ethusd:
   price of 1 ETH (USD);
3. firstCoursePrice:
   price of the first course (USD);
   set to 250 by default;

4. secondCoursePrice:
   price of the second course (USD);
   set to 500 by default;
5. firstCoursePriceWei:
   price of the first course (WEI);
6. secondCoursePriceWei:
   price of the second course (WEI);
7. minFirstCoursePriceWei:
   minimum price to be paid for the first course (90% of the price);
8. minSecondCoursePriceWei:
   minimum price to be paid for the second course (110% of the price);
9. maxFirstCoursePriceWei:
   maximum price to be paid for the first course (110% of the price);
10. maxSecondCoursePriceWei:
    maximum price to be paid for the second course (110% of the price);
11. totalPayedSumWei:
    total payment (WEI);
12. courseLevelOneBonus:
    bonus for the first level sponsor (as a percentage of the course cost);
    set to 20 by default;
13. courseLevelTwoBonus:
    bonus for a second level sponsor (as a percentage of the course price);
    set to 10 by default;
14. courseLevelThreeBonus :
    bonus for a third-level sponsor (as a percentage of the course price);
    set to 10 by default;
15. courseLevelFourBonus :
    bonus for the sponsor of the fourth level (as a percentage of the course price);
    set to 5 by default;
16. courseLevelFiveBonus:
    bonus for a level five sponsor (as a percentage of the course price);
    set to 5 by default;

17. coursePaymentsAreStopped:
    confirmation of payment acceptance activity;
    set to “false” by default;

- Note: If the price of ETH (**ethusd**) is a public variable, you should be prepared for the fact that buyers can send easily calculated 91% of the price.

Events:

1. FirstCourseBought :
   first course purchase;
2. SecondCourseBought :
   buying a second course;

The Functions:

Once contract is deployed, all variable values are set as described above.

The constructor of the contract (function **CourseContract**) executed once at creation of the contract and accepts a single parameter – the price of ETH (for example 400 $; the owner of the contract has ability to change the price with **setETHPrice** function). The constructor checks the price for correctness (must not be zero), records the contract owner as a sponsor, creates an instance of the **FundContract** contract, and runs the **pricesRecalculation** function.

The **pricesRecalculation** function recalculates variables containing the value of the courses (**firstCoursePriceWei** and **secondCoursePriceWei**) from USD to WEI.

The integration of the already published **FundContract** contract can be implemented by using the function **setFundContract** which is available only to the owner of the contract. Also when you call this function, the value of the specified course (**ethusd**) is transmitted in **FundContract**.

- Note: two functions contradict each other: the creation of instance of the **FundContract** in the **CourseContract** constructor and the **setFundContract** function which integrates the deployed **FundContract**. If the **FundContract** will be intregrated through the **setFundContract** function, the creation of instance in the constructor was unnecessary. It could be removed to save gas.

The contract contains a function that will be executed automatically when the contract is received an amount exceeding the value of the **minFirstCoursePriceWei** variable (if the **coursePaymentsAreStopped**

variable = false). The initial value of the variable **coursePaymentsAreStopped** is set to false when calling the contract constructor **CourseContract** (i.e. when it is deployed).

The value of the **coursePaymentsAreStopped** variable can be changed to true only by calling the **stopCoursePayments** function, which is available only for the owner. The value of the **coursePaymentsAreStopped** variable can be changed to false only by calling the **resumeCoursePayments** function, which is also available only for the owner.

Transcript: only owner can suspend the acceptance of payments for courses and resume it.

1. If the amount of money on the contract is less than or equal to the value of the variable **maxFirstCoursePriceWei**, the function executes the purchase of the course: **courseBought** with a parameter for the amount of funds transferred, and generates the event of the purchase of the first course - FirstCourseBought.
2. If the amount of money on the contract is less than or equal to the value of the variable **minSecondCoursePriceWei**, the difference between the amount of received funds and the value of the variable **firstCoursePriceWei** is paid to the sender, the function executes the purchase of the course: **courseBought** with a parameter of the price of the first course (**firstCoursePriceWei)**. Then, the event of the purchase of the first course (**FirstCourseBought)** is generated.
3. If the amount of money on the contract is less than or equal to the value of the variable **maxSecondCoursePriceWei**, the function executes the purchase of the course: **courseBought** with a parameter of the transferred amount. Then event **SecondCourseBought** is generated.
4. If the amount of money on the contract is greater than the value of the variable **maxSecondCoursePriceWei**, the difference between the amount of received funds and of the value of the variable **secondCoursePriceWei** is paid to the sender. The function executes the purchase of the course: **courseBought** with a parameter of the price of the first course. Then event **SecondCourseBought** is generated.

– **CourseBought function:**

Can be called only by the other functions of related contracts.

Accepts two parameters: buyer's address (buyer), payment amount (**realPayedPrice) (WEI)**.

Does the following:

1. Records the sponsor of the buyer (sponsors[buyer]) as the sponsor of the first level (sponsor1).

   Note: sponsor = referrer.

2. Sets the variable of paid bonuses (**bonusesPayed**) to 0.
3. Initializes temporary variable **bonusAmount.**
4. If the sponsor's first level is not, then records the amount of the payment for consideration to the administrator (pending), sponsors the rest of the levels simply records the owner as a sponsor.

– Note: (line 200) **sponsors[buyer]** can be changed to **sponsor1**. That variable was created for this.

5. If the first-level sponsor is the contract owner, the payment amount is transferred from the contract to the owner's address.
6. Calculates and sends bonuses to the first-level sponsor (**bonusAmount**). Then saves the amount of paid bonuses in the mapping **bonusesPayed.**
7. Performs this algorithm for the remaining 4 levels of sponsors.
8. Transfers the remaining amount (payment without paid bonuses) to the account of the contract owner.

– Note: if the sponsor of any level will be the owner of the contract, the function will try to transfer him the full amount of payment twice: in the performance of the conditional statement and at the end of the function.

– **SetSponsorInfo function:**

Can be called only by the administrator and owner.

Accepts two parameters: the address of the new participant (**newMember**) and the address of the sponsor (**sponsor**)

Does the following:

1. Checks the new sponsor for the presence of the sponsor of him.
2. If there is no new sponsor, the owner of the contract is set as new sponsor: so the corresponding bonuses is redirected to the owner.
3. Sets the new sponsor to the new participant.
4. Re-runs the **courseBought** function with the new participant's address and the amount delayed in **pending[newMember]** as parameters.
5. Runs the **newSponsorInfo** function in **fundContract** to synchronize the sponsorship system in two contracts.

– **TestSetSponsorInfo function:**

Can be called only by the administrator and owner.

Accepts two parameters: the address of the new participant (**newMember**) and the address of the sponsor (**sponsor**)

Does the following:

1. Runs the **newSponsorInfo** function in the **fundContract.**

– **GetSponsor function:**

Accepts single parameter: the address of the participant.
Does not require a sending of the transaction and gas.
Does the following:

1. Returns the address of the first-level sponsor.

– **SetETHPrice function:**

Can be called only by the owner.
Accepts single parameter: new price of the ETH.
Does the following:

1. Checks that the new price is greater than 0.
2. Sets a new price.
3. Sets new price in the **fundContract.**
4. Runs the recalculation of the prices (**pricesRecalculation**)

– **setFirstCoursePrice function:**

Can be called only by the owner.
Accepts single parameter: new price of the first course.
Does the following:

1. Checks that the new price is greater than 0 and less than 5000.
2. Changes the price of the first course to a new one.
3. Runs the recalculation of the prices (**pricesRecalculation**).

– **setSecondCoursePrice function:**

Can be called only by the owner.
Accepts single parameter: new price of the second course.
Does the following:

1. Checks that the new price is greater than 0 and less than 5000.
2. Changes the price of the second course to a new one.
3. Runs the recalculation of the prices (pricesRecalculation).

– **stopCoursePayments function:**

Can be called only by the owner.
Does the following:

1. Sets **coursePaymentsAreStopped** to true. It means that users can no longer send money to the contract.

– **resumeCoursePayments function:**

Can be called only by the owner.
Does the following:

1. Sets **coursePaymentsAreStopped** to false. It means that users can send money to the contract.

2. FundContract

The declared functionality of the **FundContract** contract is the acceptance of deposits in ETH. The Interest is paid during the Deposit Term. The amount of deposit is paid out at the end of Deposit Term.

Contract inherits from contracts **Adminable** and **Ownable**.

The functionality of the contract **Adminable** implements the access restriction to functions of the contract by the modifier **onlyAdminAndOwner**.

The modifier **onlyAdminAndOwner** restricts the calling functions with roles owner or admin. The admin role can be set via the **setAdmin** function, which is available only to the owner role (via the **onlyOwner** modifier implemented in the **Ownable** contract).

The functionality of the contract **Ownable** implements the access restriction functions of the contract by the modifier **onlyOwner**.

Modifier **onlyOwner** restricts the function call with the owner role. The owner role is set when the contract is deployed (with the value of the address which deployed it). The new owner value can be set by calling the **transferOwnership** function through the address **newOwner** parameter. The **newOwner** value cannot be zero.

Variables

1. name = "BitWork Fund Conrtact":
   title;
   - Note: possible typo in the word “Contract”;
2. CourseContract public courseContract:
   contract **courseContract**;
3. ethusd:
   price of the ETH (USD);
4. mapping (address => uint) public pending:
   pended deposits for each address;
5. fundOneInvestorPersent = 50:
   50% - cumulative amount of Interest at the end of the Deposit Term at for the first type of deposit.

6. fundOneSponsorOnePersent = 4:
   additional Interest for the first-level sponsor in the first type of deposit;
7. fundOneSponsorTwoPersent = 3:
   additional Interest for the second-level sponsor in the first type of deposit;
8. fundOneSponsorThreePersent = 3:
   additional Interest for the third-level sponsor in the first type of deposit;
9. fundOneSponsorFourPersent = 0:
   additional Interest for the fourth-level sponsor in the first type of deposit;
10. fundTwoInvestorPersent = 100:
    100% - cumulative amount of Interest at the end of the Deposit Term at for the second type of deposit.

    – Problem: the second tariff is economically less profitable, because it is possible to invest twice in a row for 6 months each time for profit to be 125%. Users will not apply to the second type deposit.

11. fundTwoSponsorOnePersent = 8:
    additional Interest for the first-level sponsor in the second type of deposit;
12. fundTwoSponsorTwoPersent = 6:
    additional Interest for the second -level sponsor in the second type of deposit;
13. fundTwoSponsorThreePersent = 3:
    additional Interest for the third -level sponsor in the second type of deposit;
14. fundTwoSponsorFourPersent = 3:
    additional Interest for the fourth -level sponsor in the second type of deposit;

    – Note: some of the variables have the same value. It is possible to join them, since there is no function to change them.

15. totalPaymentsUSD = 0:
    final amount of payments to users (USD);
16. totalPaymentsWei = 0:
    final amount of payments to users (WEI);
17. totalInvestmentsUSD = 0:
    final amount of investments (USD);
18. totalInvestmentsWei = 0:
    final amount of investments (WEI);
19. maxFirstFundAmountUSD = 10000:
    USD limit for the first type deposits;

– Note: users can can easily get around limit of the first type deposits by splitting a large amount into smaller amounts and making deposits from different Ethereum anonymous addresses.

20. divisionInaccuracyProtection = 20:
    Inaccuracy of the division;
21. period = 30 days:
    time period to divide the term (for paying Interest monthly)
22. fundPaymentsAreStopped = false:
    status of the system (accepting payments: false – yes, true – no)

23. testTimeShift = 0:
    the test variable for shift time of "now"

    – Problem: since the contract has a function **setTestTimeShift,** timeshift is recommended to be removed to eliminate the possibility of interference to the system.

24. Entry:
    struct that is supposed to hold the address of the participant (**person**), the Deposit amount (**value**), the total Interest at the end of the term (**persent**), the type of the deposit (1 or 2) (**fund**), whether the investor is a participant (true or false) (**isInvestor**), the time of the Deposit (**startTime**), how much has already been withdrawn (USD) (**withdrawn**).

    – Note: one parameter has a grammatical error (**persent**): correctly – percent.

The Functions:

Once contract is deployed, all variable values are set as described above.
Also an **Entry** array called deposits is created.

The constructor of the contract (function **FundContract**) is executed once at creation of the contract.
It accepts one parameter: address of the contract **courseContract.**
It allows **courseContract** and **FundContract** integrate.

– **NewDeposit function:**

Accepts 6 parameters:

1. The address of the participant (**person**);
2. The value of the Deposit (**value**);
3. Total Interest at the end of the term (**persent**);

4. The type of the deposit (1 or 2) (**fund**);
5. Whether a participant is an investor (true or false) (**isInvestor**);
6. Time of the acceptance of the deposit (**startTime**);

Does the following:

1. Creates a new Entry struct in the **deposits** array;

– **Fallback function**:

(automatically enabled when contract accepts money)

Does the following:

1. Checks the price of ETH (**ethusd** must be above 0).
2. If the owner/administrator is the sender of the money, function calculates how much money contract has for payments, and if enough, it calls the **makePayments** function.
3. In other cases, it checks if the acceptance of payments is not suspended (**fundPaymentsAreStopped**). Then amount is sent to the owner as well as amount of the final investment is stored.
4. Requests information about the sponsor from the **courseContract**.

   – Note: the excessive functionality: createDepositRecords() (paragraph 6) will do the same operation

5. If there is no sponsor, the amount is recorded as **pending.**
6. Calls **CreateDepositRecords** function with parameters of the sender's address and the amount of money received (only number, not real money);

– **createDepositRecords function:**

– Note: title of the function as **"createDepositRecord"** is more appropriate by its meaning.

Can be called only by the other functions of related contracts.
Accepts two parameters: the address of the investor, the investment amount.

Does the following:

1. Requests information about the sponsor from the **courseContract.**

2. If the payment is less than the maximum for the first type deposit:
   - Creates an investor's struct containing all information (by calling the already described **newDeposit** function with 6 parameters)
   - Checks the presence of the first-level sponsor (if there is no such sponsor, the owner becomes sponsor)
   - Then creates the corresponding struct for the sponsor (by calling the already described newDeposit function with 6 parameters)
   - Checks the presence of sponsors of other levels (in order) and implements the same algorithm.

     - Note: error in code (line 467) initialization of the third-level sponsor should be two rows above.

     - Note: function checks the presence of the fourth-level sponsor which according the rules will receive nothing.

3. If the payment is greater than the maximum for the first type deposit then the above algorithm is done to record the payment for the second type deposit.

   – Problem: for level 2, 3 and 4 sponsors, the same error of changing row locations is made, i.e. the owner is set as a sponsor under a certain condition, and then this variable is requested from another contract, which makes the previous action meaningless

   – Problem: the logic of the above function divides tariff payments accordingly to the amount of money, not on the customer's preference, which makes it impossible to call this function correctly if the customer wants to invest more than the maximum of the first tariff for a period of 6 months, and Vice versa, if the customer wants to invest a small amount for 12 months.

– **makePayments function:**

Can be called only by the other functions of related contracts.
Does not accept parameters.

Does the following:

1. Calculates the amount to be paid at the moment for each deposit record (by calling the **totalPersonPaymentsShouldBeDoneTothemoment** function)
2. If possible executes the payment.

– Note: error in code (line 515): the **totalPersonPaymentsShouldBeDoneTothemoment** function is called with a space between the function name and parentheses with parameters.

– **getAmountToPayAtTheMoment function:**

Accepts single parameter: time.
Can be called only by the administrator and owner.
Does not require a sending transaction and gas.
Returns the amount to be paid (number, not money).

Does the following:

1. Checks the specified time for correctness for this function (it is impossible to take the past period, i.e. time >= now);
2. Calculates amount to be paid at a given point for each record in the **deposits** array (by calling the function **totalPersonPaymentsShouldBeDoneTothemoment**);
3. Returns the amount to be paid (number, not money);

– **totalPersonPaymentsShouldBeDoneTothemoment function:**

Accepts two arguments: index of the desired record in the deposits array, time.
Does not require a sending transaction and gas.
Returns the amount to be paid (number, not money).

Does the following:

1. Checks the specified time for correctness for this function (it is impossible to take the past period, i.e. time >= now)
2. Calculates amount to be paid at a given point for desired record in the **deposits** array

– Note: presumably typos:
Total Interest of the first type deposit is equal to 100% (should be 50).
Month Interest of the first type deposit is equal to 20% (should be 10).
Formula of the total amount to be paid contains interest twice in a row.
– Note: one of the variables is called monthes (grammatical error, must be months).

– **getAmountToPayNow function:**

Does not accept parameters.
Does not require a sending transaction and gas.
Can be called only by the administrator and owner.
Returns the amount to be paid (number, not money).
Does the following:

1. Calls the getAmountToPayAtTheMoment function with the parameter of time "Now".

– **getAmountToPayTommorow function:**

Does not accept parameters.
Does not require a sending transaction and gas.
Can be called only by the administrator and owner.
Returns the amount to be paid (number, not money).
Does the following:

1. Calls the getAmountToPayAtTheMoment function with the time parameter "Now + 1 days" .

– **getAmountToPayNextWeek function:**

Does not accept parameters.
Does not require a sending transaction and gas.
Can be called only by the administrator and owner.
Returns the amount to be paid (number, not money).
Does the following:

1. Calls the getAmountToPayAtTheMoment function with the time parameter "Now + 7 days" .

– **getAmountToPayNextMonth function:**

Does not accept parameters.
Does not require a sending transaction and gas.
Can be called only by the administrator and owner.
Returns the amount to be paid (number, not money).
Does the following:

1. Calls the getAmountToPayAtTheMoment function with the time parameter "Now + 30 days"

– **getContractBalanceWei function:**

Does not accept parameters.
Does not require a sending transaction and gas.
Does the following:

1. Returns the balance of the contract (WEI).

– **getContractBalanceUSD function:**

Does not accept parameters.
Does not require a transaction and gas.
Does the following:

1. Returns the balance of the contract (USD).

– **withdraw function:**

Does not accept parameters
Can be called only by the owner.
Does the following:

1. Transfers the whole balance of the contract to the owner.

– **getTotalInvestmentsUSD function:**

Does not accept parameters.
Does not require a transaction and gas.
Can be called only by the administrator and owner.
Returns total investments (USD) (number, not money).

– **getTotalInvestmentsWei function:**

Does not accept parameters.
Does not require a transaction and gas.
Can be called only by the administrator and owner.
Returns total investments (WEI) (number, not money).

- **getTotalPaymentsUSD function:**

  Does not accept parameters.
  Does not require a transaction and gas.
  Can be called only by the administrator and owner.
  Returns the sum of payments to investors (USD) (number, not money).

- **getTotalPaymentsWei function:**

  Does not accept parameters.
  Does not require a transaction and gas.
  Can be called only by the administrator and owner.
  Returns the sum of payments to investors (WEI) (number, not money).

- **setETHPrice function:**

  Accepts single parameter: new price of ETH.
  Can be called only from the **courseContract**.
- Note: it turns out it could be internal.
  Changes the price of ETH.

    - Note: the function does not trigger any recalculations, thus all USD values saved in the contract are outdated after the price updating.

- **toWei function:**

  Accepts single parameter: the number of usd (number, not money).
  Does not require a sending transaction and gas.
  Returns calculated amount in Wei (number, not money).

- **toUSD function:**

  Accepts single parameter: the number of WEI (number, not money).
  Does not require a sending transaction and gas.
  Returns calculated amount in USD(number, not money)

- **newSponsorInfo function:**

Accepts single parameter: the address of the new member.
Can be called only from the **courseContract**.

– Note: it turns out it could be internal.
Does the following:

1. Runs **createDepositRecords** function that creates the sponsor record in the deposits array.

– **stopFundPayments function:**

Does not accept parameters.
Can be called only by the owner.
Does the following:
Changes the value of the **FundPaymentsAreStopped** to “true” (makes users unable to send money to the contract)

– **resumeFundPayments function:**

Does not accept parameters.
Can be called only by the owner.
Does the following:
Changes the value of the **FundPaymentsAreStopped** to “false” (makes users able to send money to the contract)

– **setCourseContract function:**

Accepts single parameter: the address of the new **courseContract** contract.
Can be called only by the owner.
Does the following:
Integrates a new **courseContract** contract.

– **setTestTimeShift function:**

Accepts single parameter: time-shift value (added to “now”, for test purposes)
Can be called only by the owner.
Does the following:

Sets a new time-shift value.

- Note: time-shift is recommended to be removed to eliminate the possibility of interference to the system.

Disclaimer of Liability

This audit concerns only the source codes of smart contracts and should not be regarded as the approval of a platform, team or company.

Authors

The audit was conducted by EthereumWorks team. For issues related to conducting audits and developing smart contracts please contact: - @SlavaPoe (Telegram) - v.poskonin, MousePo (Skype).