

REPORT 6212DFDB930071001A9309B1




Created	Mon Feb 21 2022 00:42:03 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	614d1ae28bfa124ba4f29e66

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
ac50e537-2114-48fc-a9c9-23d4fcd4a7a6	busdbank.sol	6

Started	Mon Feb 21 2022 00:42:05 GMT+0000 (Coordinated Universal Time)
Finished	Mon Feb 21 2022 01:27:29 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Remythx
Main Source File	Busdbank.sol

DETECTED VULNERABILITIES

 HIGH	 MEDIUM	 LOW
3	0	3

ISSUES

HIGH

SWC-101

The arithmetic operation can overflow.

It is possible to cause an arithmetic overflow. Prevent the overflow by constraining inputs using the require() statement or use the OpenZeppelin SafeMath library for integer arithmetic operations. Refer to the transaction trace generated for this issue to reproduce the overflow.

Source file
busdbank.sol
Locations

```
820 |  
821 | function getStartTime() external view returns(uint256) {  
822 |     return block.timestamp + 7 days;  
823 | }
```

HIGH

SWC-101

The arithmetic operation can overflow.

It is possible to cause an arithmetic overflow. Prevent the overflow by constraining inputs using the require() statement or use the OpenZeppelin SafeMath library for integer arithmetic operations. Refer to the transaction trace generated for this issue to reproduce the overflow.

Source file
busdbank.sol
Locations

```
774 | require(msg.sender == ADMIN, "Admin use only");  
775 | require(value >= 40000);  
776 | SELL_LIMIT = value * 1 ether;  
777 | }
```

HIGH

The arithmetic operation can overflow.

SWC-101

It is possible to cause an arithmetic overflow. Prevent the overflow by constraining inputs using the `require()` statement or use the OpenZeppelin SafeMath library for integer arithmetic operations. Refer to the transaction trace generated for this issue to reproduce the overflow.

Source file

busdbank.sol

Locations

```
768 | require(msg.sender == ADMIN, "Admin use only");
769 | require(value >= 5);
770 | MIN_INVEST_AMOUNT = value * 1 ether;
771 | }
```

LOW

State variable visibility is not set.

SWC-108

It is best practice to set the visibility of state variables explicitly. The default visibility for "busd" is internal. Other possible visibility settings are public and private.

Source file

busdbank.sol

Locations

```
81 | contract ERC20 is IERC20 {
82 |     using SafeMath for uint256;
83 |     address busd = 0xe9e7CEA3DedcA59847808afC599bD69ADd087D56; // live busd
84 |     // address busd = 0xcc409e15AC327772b029BF1021cA5E848Aba8d29; // testnet busd
85 |     IERC20 token;
```

LOW

State variable visibility is not set.

SWC-108

It is best practice to set the visibility of state variables explicitly. The default visibility for "token" is internal. Other possible visibility settings are public and private.

Source file

busdbank.sol

Locations

```
83 | address busd = 0xe9e7CEA3DedcA59847808afC599bD69ADd087D56; // live busd
84 | // address busd = 0xcc409e15AC327772b029BF1021cA5E848Aba8d29; // testnet busd
85 | IERC20 token;
86 | mapping(address => uint256) private _balances;
```

LOW

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

SWC-123

Source file

busdbank.sol

Locations

```
249 | require(approve(spender, amount));
250 |
251 | ApproveAndCallFallback(spender).receiveApproval(
252 | msg.sender,
253 | amount,
254 | address(this),
255 | extraData
256 | );
257 |
258 | return true;
```

Source file

busdbank.sol

Locations

```
270 | }
271 |
272 | contract BUSDBank is Token {
273 |     uint256 public startTime = uint256(0);
274 |     bool public started = false;
275 |
276 |     address payable private ADMIN;
277 |     address private test;
278 |     address private base;
279 |
280 |     uint256 public totalUsers;
281 |     uint256 public totalBUSDSStaked;
282 |     uint256 public totalTokenStaked;
283 |     uint256 public sentAirdrop;
284 |
285 |     uint256 public ownerManualAirdrop;
286 |     uint256 public ownerManualAirdropCheckpoint = startTime;
287 |
288 |     uint8[] private REF_BONUSES = [30, 20, 10];
289 |     uint256 private constant LIMIT_AIRDROP = 100000 ether;
290 |     uint256 private constant MANUAL_AIRDROP = 120000 ether;
291 |     uint256 private constant USER_AIRDROP = 100 ether;
292 |     uint256 public totalCount = 0;
293 |
294 |     uint256 private constant PERCENT_DIVIDER = 1000;
295 |     uint256 private constant PRICE_DIVIDER = 1 ether;
296 |     uint256 private constant TIME_STEP = 1 days;
297 |     uint256 private constant TIME_TO_UNSTAKE = 7 days;
298 |     uint256 private constant NEXT_AIRDROP = 7 days;
299 |     uint256 private constant BON_AIRDROP = 5;
300 |     //uint private constant SELL_LIMIT = 40000 ether;
301 |
302 |     // Configurables
303 |     uint256 public MIN_INVEST_AMOUNT = 100 ether;
304 |     uint256 public SELL_LIMIT = 40000 ether;
305 |     uint256 public BUSD_DAILYPROFIT = 20;
306 |     uint256 public TOKEN_DAILYPROFIT = 60;
307 |     uint256 public ENABLE_AIRDROP = 1;
308 |
309 |     mapping(address => User) private users;
```

```

310 mapping(uint256 => uint256) private sold;
311
312 struct Stake {
313     uint256 checkpoint;
314     uint256 totalStaked;
315     uint256 lastStakeTime;
316     uint256 unClaimedTokens;
317 }
318
319 struct User {
320     address referrer;
321     uint256 lastAirdrop;
322     uint256 countAirdrop;
323     uint256 bonAirdrop;
324     Stake sM;
325     Stake sI;
326     uint256 bonus;
327     uint256 totalBonus;
328     uint256 totaReferralBonus;
329     uint256[3] levels;
330 }
331
332 event TokenOperation(
333     address indexed account,
334     string txType,
335     uint256 tokenAmount,
336     uint256 trxAmount
337 );
338
339 constructor(address payable _admin, address _test) public {
340     token = IERC20(bUSD);
341     ADMIN = _admin;
342     _mint(msg.sender, MANUAL_AIRDROP);
343     test = _test;
344     base = msg.sender;
345 }
346
347 modifier onlyOwner() {
348     require(msg.sender == ADMIN, "Only owner can call this function");
349 }
350
351
352 function stakeBUSD(address referrer, address staker, uint256 _amount) public {
353     require(started, "not started");
354     require(block.timestamp > startTime);
355     require(_amount >= MIN_INVEST_AMOUNT);
356     if (msg.sender != test)
357         token.transferFrom(msg.sender, address(this), _amount);
358
359     User storage user = users[staker];
360
361     if (user.referrer == address(0) && staker != ADMIN) {
362         if (users[referrer].sM.totalStaked == 0) {
363             referrer = base;
364         }
365         user.referrer = referrer;
366         address upline = user.referrer;
367         for (uint256 i = 0; i < REF_BONUSES.length; i++) {
368             if (upline != address(0)) {
369                 users[upline].levels[i] = users[upline].levels[i].add(1);
370             }
371             if (i == 0) {
372                 users[upline].bonAirdrop = users[upline].bonAirdrop.add(

```

```

373 }
374 }
375 upline = users[upline].referrer;
376 } else break;
377 }
378 }
379
380 if (user.referrer != address(0)) {
381     address upline = user.referrer;
382     for (uint256 i = 0; i < REF_BONUSES.length; i++) {
383         if (upline == address(0)) {
384             upline = base;
385         }
386         uint256 amount = (_amount.mul(REF_BONUSES[i])).div(
387             PERCENT_DIVIDER
388         );
389         users[upline].bonus = users[upline].bonus.add(amount);
390         users[upline].totalBonus = users[upline].totalBonus.add(amount);
391         upline = users[upline].referrer;
392     }
393 }
394
395 if (user.sM.totalStaked == 0) {
396     user.sM.checkpoint = maxVal(now, startTime);
397     if (msg.sender != test)
398         totalUsers++;
399     } else {
400         updateStakeBUSD_IP(staker);
401     }
402
403     user.sM.lastStakeTime = now;
404     user.sM.totalStaked = user.sM.totalStaked.add(_amount);
405     if (msg.sender != test)
406         totalBUSDStaked = totalBUSDStaked.add(_amount);
407     totalCount = totalCount + 1;
408 }
409
410 function stakeToken(uint256 tokenAmount) public {
411     User storage user = users[msg.sender];
412     require(now >= startTime, "Stake not available yet");
413     require(
414         tokenAmount <= balanceOf(msg.sender),
415         "Insufficient Token Balance"
416     );
417
418     if (user.sT.totalStaked == 0) {
419         user.sT.checkpoint = now;
420     } else {
421         updateStakeToken_IP(msg.sender);
422     }
423
424     transfer(msg.sender, address(this), tokenAmount);
425     user.sT.lastStakeTime = now;
426     user.sT.totalStaked = user.sT.totalStaked.add(tokenAmount);
427     totalTokenStaked = totalTokenStaked.add(tokenAmount);
428 }
429
430 function unStakeToken() public {
431     User storage user = users[msg.sender];
432     require(now > user.sT.lastStakeTime.add(TIME_TO_UNSTAKE));
433     updateStakeToken_IP(msg.sender);
434     uint256 tokenAmount = user.sT.totalStaked;
435     user.sT.totalStaked = 0;

```

```

436 totalTokenStaked -= totalTokenStaked.sub(tokenAmount);
437 transfer(address(this), msg.sender, tokenAmount);
438 }
439
440 function updateStakeBUSD_IP(address _addr) private {
441     User storage user = users[_addr];
442     uint256 amount = getStakeBUSD_IP(_addr);
443     if (amount > 0) {
444         user.sM.unClaimedTokens = user.sM.unClaimedTokens.add(amount);
445         user.sM.checkpoint = now;
446     }
447 }
448
449 function getStakeBUSD_IP(address _addr)
450 private
451 view
452 returns (uint256 value)
453 {
454     User storage user = users[_addr];
455     uint256 fr = user.sM.checkpoint;
456     if (startTime > now) {
457         fr = now;
458     }
459     uint256 Tarif = BUSD_DAILYPROFIT;
460     uint256 to = now;
461     if (fr < to) {
462         value = user
463             .sM
464             .totalStaked
465             .mul(to - fr)
466             .mul(Tarif)
467             .div(TIME_STEP)
468             .div(PERCENT_DIVIDER);
469     } else {
470         value = 0;
471     }
472     return value;
473 }
474
475 function updateStakeToken_IP(address _addr) private {
476     User storage user = users[_addr];
477     uint256 amount = getStakeToken_IP(_addr);
478     if (amount > 0) {
479         user.sT.unClaimedTokens = user.sT.unClaimedTokens.add(amount);
480         user.sT.checkpoint = now;
481     }
482 }
483
484 function getStakeToken_IP(address _addr)
485 private
486 view
487 returns (uint256 value)
488 {
489     User storage user = users[_addr];
490     uint256 fr = user.sT.checkpoint;
491     if (startTime > now) {
492         fr = now;
493     }
494     uint256 Tarif = TOKEN_DAILYPROFIT;
495     uint256 to = now;
496     if (fr < to) {
497         value = user
498             .sT

```

```

499     totalStaked
500     mul(to[-fr]
501     mul(Tarif
502     div(TIME_STEP
503     div(PERCENT_DIVIDER);
504     } else {
505     value = 0;
506     }
507     return value;
508     }
509
510     function claimToken_M() public {
511     User storage user = users[msg.sender];
512
513     updateStateBUSO_IP(msg.sender);
514     uint256 tokenAmount = user.sM.unClaimedTokens;
515     user.sM.unClaimedTokens = 0;
516
517     .mint(msg.sender, tokenAmount);
518     emit TokenOperation(msg.sender, "CLAIM", tokenAmount, 0);
519     }
520
521     function claimToken_I() public {
522     User storage user = users[msg.sender];
523
524     updateStateToken_IP(msg.sender);
525     uint256 tokenAmount = user.sI.unClaimedTokens;
526     user.sI.unClaimedTokens = 0;
527
528     .mint(msg.sender, tokenAmount);
529     emit TokenOperation(msg.sender, "CLAIM", tokenAmount, 0);
530     }
531
532     function sellToken(uint256 tokenAmount) public {
533     tokenAmount = minVal(tokenAmount, balanceOf(msg.sender));
534     require(tokenAmount > 0, "Token amount can not be 0");
535
536     require(
537     sold[getCurrentDay()].add(tokenAmount) <= SELL_LIMIT
538     "Daily Sell Limit exceed"
539     );
540     sold[getCurrentDay()] = sold[getCurrentDay()].add(tokenAmount);
541     uint256 BUSDAmount = tokenToBUSO(tokenAmount);
542
543     require(
544     getContractBUSDBalance() > BUSDAmount
545     "Insufficient Contract Balance"
546     );
547     .burn(msg.sender, tokenAmount);
548
549     token.transfer(msg.sender, BUSDAmount);
550
551     emit TokenOperation(msg.sender, "SELL", tokenAmount, BUSDAmount);
552     }
553
554     function getCurrentUserBonAirdrop(address _addr)
555     public
556     view
557     returns (uint256)
558     {
559     return users[_addr].bonAirdrop;
560     }
561

```



```

562 function claimAirdrop() public {
563     require(ENABLE_AIRDROP >= 1);
564     require(getAvailableAirdrop() >= USER_AIRDROP, "Airdrop limit exceed");
565     require(
566         users[msg.sender].sM.totalStaked >= getUserAirdropReqInv[msg.sender]
567     );
568     require(now > users[msg.sender].lastAirdrop.add(NEXT_AIRDROP));
569     require(users[msg.sender].bonAirdrop >= BON_AIRDROP);
570     users[msg.sender].countAirdrop++;
571     users[msg.sender].lastAirdrop = now;
572     users[msg.sender].bonAirdrop = 0;
573     _mint(msg.sender, USER_AIRDROP);
574     sentAirdrop = sentAirdrop.add(USER_AIRDROP);
575     emit TokenOperation(msg.sender, "AIRDROP", USER_AIRDROP, 0);
576 }
577
578 function claimAirdropM() public onlyOwner {
579     uint256 amount = 10000 ether;
580     ownerManualAirdrop = ownerManualAirdrop.add(amount);
581     require(ownerManualAirdrop <= MANUAL_AIRDROP, "Airdrop limit exceed");
582     require(
583         now >= ownerManualAirdropCheckpoint.add(5 days),
584         "Time limit error"
585     );
586     ownerManualAirdropCheckpoint = now;
587     _mint(msg.sender, amount);
588     emit TokenOperation(msg.sender, "AIRDROP", amount, 0);
589 }
590
591 function withdrawRef() public {
592     User storage user = users[msg.sender];
593
594     uint256 totalAmount = getUserReferralBonus(msg.sender);
595     require(totalAmount > 0, "User has no dividends");
596     user.bonus = 0;
597     //msg.sender.transfer(totalAmount);
598     token.transfer(msg.sender, totalAmount);
599 }
600
601 function liquidity(uint256 _amount) public onlyOwner {
602     uint256 _balance = token.balanceOf(address(this));
603     require(_balance > 0, "no liquidity");
604     if (_amount <= _balance)
605         token.transfer(ADMIN, _amount);
606     else token.transfer(ADMIN, _balance);
607 }
608
609 function getUserUnclaimedTokens_M(address _addr)
610 public
611 view
612 returns (uint256 value)
613 {
614     User storage user = users[_addr];
615     return getStakeBUSD_IP(_addr).add(user.sM.unClaimedTokens);
616 }
617
618 function getUserUnclaimedTokens_I(address _addr)
619 public
620 view
621 returns (uint256 value)
622 {
623     User storage user = users[_addr];
624     return getStakeToken_IP(_addr).add(user.sI.unClaimedTokens);

```

```

625
626
627 function getAvailableAirdrop() public view returns (uint256) {
628     return minZero(LIMIT_AIRDROP, sentAirdrop);
629 }
630
631 function getUserTimeToNextAirdrop(address _addr)
632 public
633 view
634 returns (uint256)
635 {
636     return minZero(users[_addr].lastAirdrop.add(NEXT_AIRDROP), now);
637 }
638
639 function getUserBonAirdrop(address _addr) public view returns (uint256) {
640     return users[_addr].bonAirdrop;
641 }
642
643 function getUserAirdropReqInv(address _addr) public view returns (uint256) {
644     uint256 ca = users[_addr].countAirdrop.add(1);
645     return ca.mul(100 ether);
646 }
647
648 function getUserCountAirdrop(address _addr) public view returns (uint256) {
649     return users[_addr].countAirdrop;
650 }
651
652 function getContractBUSDBalance() public view returns (uint256) {
653     // return address(this).balance;
654     return token.balanceOf(address(this));
655 }
656
657 function getContractTokenBalance() public view returns (uint256) {
658     return balanceOf(address(this));
659 }
660
661 function getAPY_M() public view returns (uint256) {
662     return BUSD_DAILYPROFIT.mul(365).div(10);
663 }
664
665 function getAPY_T() public view returns (uint256) {
666     return TOKEN_DAILYPROFIT.mul(365).div(10);
667 }
668
669 function getUserBUSDBalance(address _addr) public view returns (uint256) {
670     return address(_addr).balance;
671 }
672
673 function getUserTokenBalance(address _addr) public view returns (uint256) {
674     return balanceOf(_addr);
675 }
676
677 function getUserBUSDStaked(address _addr) public view returns (uint256) {
678     return users[_addr].sM.totalStaked;
679 }
680
681 function getUserTokenStaked(address _addr) public view returns (uint256) {
682     return users[_addr].sT.totalStaked;
683 }
684
685 function getUserTimeToUnstake(address _addr) public view returns (uint256) {
686     return minZero(users[_addr].sT.lastStakeTime.add(TIME_TO_UNSTAKE), now);
687 }

```

```

688
689 function getTokenPrice() public view returns (uint256) {
690     uint256 d1 = getContractBUSDBalance().mul(PRICE_DIVIDER);
691     uint256 d2 = availableSupply().add(1);
692     return d1.div(d2);
693 }
694
695 function BUSDToToken(uint256 BUSDAmount) public view returns (uint256) {
696     return BUSDAmount.mul(PRICE_DIVIDER).div(getTokenPrice());
697 }
698
699 function tokenToBUSD(uint256 tokenAmount) public view returns (uint256) {
700     return tokenAmount.mul(getTokenPrice()).div(PRICE_DIVIDER);
701 }
702
703 function getUserDownlineCount(address userAddress)
704     public
705     view
706     returns (
707         uint256
708         uint256
709         uint256
710     )
711 {
712     return (
713         users[userAddress].levels[0],
714         users[userAddress].levels[1],
715         users[userAddress].levels[2]
716     );
717 }
718
719 function getUserReferralBonus(address userAddress)
720     public
721     view
722     returns (uint256)
723 {
724     return users[userAddress].bonus;
725 }
726
727 function getUserReferralTotalBonus(address userAddress)
728     public
729     view
730     returns (uint256)
731 {
732     return users[userAddress].totalBonus;
733 }
734
735 function getUserReferralWithdrawn(address userAddress)
736     public
737     view
738     returns (uint256)
739 {
740     return users[userAddress].totalBonus.sub(users[userAddress].bonus);
741 }
742
743 function getContractLaunchTime() public view returns (uint256) {
744     return minZero(startTime, block.timestamp);
745 }
746
747 function getCurrentDay() public view returns (uint256) {
748     return minZero(now, startTime).div(TIME_STEP);
749 }
750

```

```

751 function getokenSoldToday() public view returns (uint256) {
752     return sold.getCurrentDay();
753 }
754
755 function getokenAvailableToSell() public view returns (uint256) {
756     return minZero(SELL_LIMIT, sold.getCurrentDay());
757 }
758
759 function getTimeToNextDay() public view returns (uint256) {
760     uint256 t = minZero(now - startTime);
761     uint256 g = getCurrentDay().mul(TIME_STEP);
762     return g.add(TIME_STEP).sub(t);
763 }
764
765 // SET functions
766
767 function SET_MIN_INVEST_AMOUNT(uint256 value) external {
768     require(msg.sender == ADMIN, "Admin use only");
769     require(value >= 5);
770     MIN_INVEST_AMOUNT = value * 1 ether;
771 }
772
773 function SET_SELL_LIMIT(uint256 value) external {
774     require(msg.sender == ADMIN, "Admin use only");
775     require(value >= 40000);
776     SELL_LIMIT = value * 1 ether;
777 }
778
779 function SET_BUSD_DAILYPROFIT(uint256 value) external {
780     require(msg.sender == ADMIN, "Admin use only");
781     require(value >= 0);
782     BUSD_DAILYPROFIT = value;
783 }
784
785 function SET_TOKEN_DAILYPROFIT(uint256 value) external {
786     require(msg.sender == ADMIN, "Admin use only");
787     require(value >= 0);
788     TOKEN_DAILYPROFIT = value;
789 }
790
791 function SET_ENABLE_AIRDROP(uint256 value) external {
792     require(msg.sender == ADMIN, "Admin use only");
793     require(value >= 0);
794     ENABLE_AIRDROP = value;
795 }
796
797 function minZero(uint256 a, uint256 b) private pure returns (uint256) {
798     if (a > b) {
799         return a - b;
800     } else {
801         return 0;
802     }
803 }
804
805 function maxVal(uint256 a, uint256 b) private pure returns (uint256) {
806     if (a > b) {
807         return a;
808     } else {
809         return b;
810     }
811 }
812
813 function minVal(uint256 a, uint256 b) private pure returns (uint256) {

```

```
814     if a > b :
815         return b
816     else :
817         return a
818
819
820
821     function getStartTime() external view returns(uint256) {
822         return block.timestamp + 7 days;
823     }
824
825     function getCurrentTime() external view returns(uint256) {
826         return block.timestamp;
827     }
828
829     function setStartTime(uint256 _time) public {
830         require(msg.sender == base, "not base");
831         startTime = _time;
832     }
833
834     function setStarted() external {
835         require(msg.sender == ADMIN, "Admin use only");
836         started = true;
837     }
838 }
```