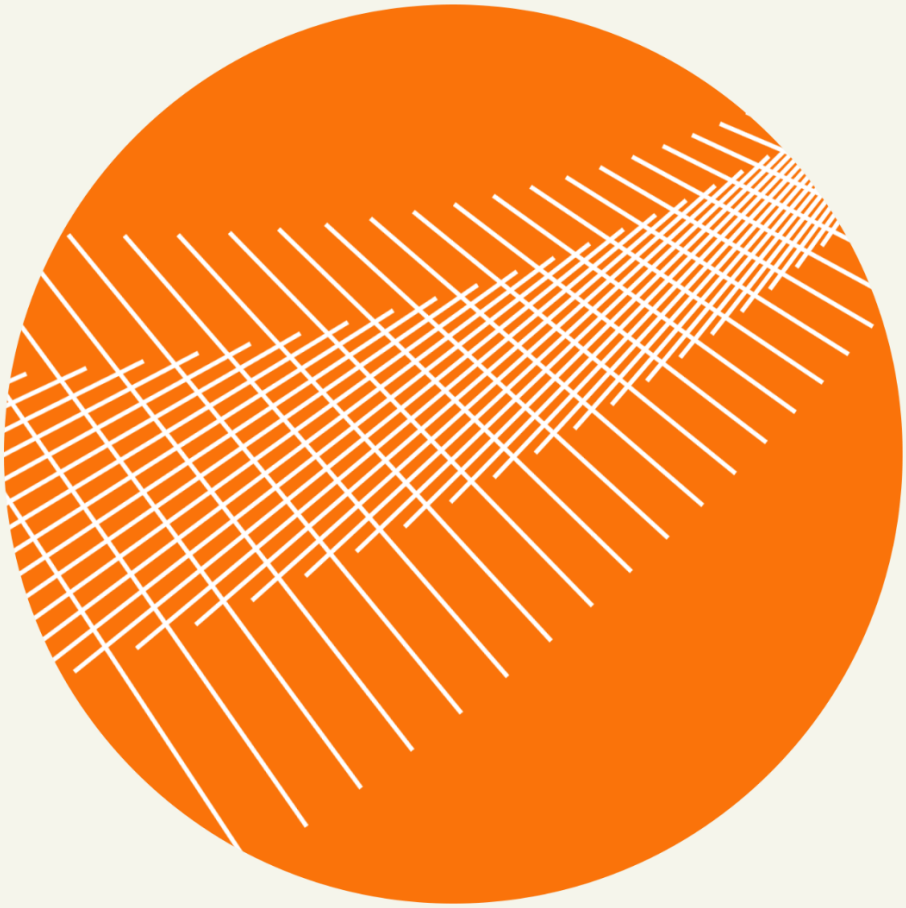
CTDSEC

YOU ARE PROTECTED



TERMS

No part of this publication, in whole or in part, may be reproduced, copied, transferred or any other right reserved to its copyright a CTDSec, including photocopying and all other copying, any transfer or transmission using any network or other means of communication, in any form or by any means such as any information storage, transmission or retrieval system, without prior written permission.

Smart contract security audit

BurnItDAO

Table of Contents

1.0 Introduction **3**
1.1 Project engagement 3
1.2 Disclaimer 3
2.0 Coverage **4**
2.1 Target Code and Revision 4
2.2 Attacks made to the contract 5
3.0 Security Issues **7**
3.1 High severity issues 7
3.2 Medium severity issues 7
3.3 Low severity issues [1 - Mitigated] 7
3.3 Informational Findings [1 - Not apply] 8
4.0 Testing coverage - python **9**
5.0 Annexes **11**
6.0 Summary of the audit **22**

Copyright © ***CTDSec*** - All rights reserved

1.0 Introduction

1.1 Project engagement

During December of 2023, BurnItDAO team engaged CTDSec to audit smart contracts that they created. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts. BurnItDAO provided CTDSec with access to their code repository and whitepaper.

1.2 Disclaimer

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that's able to withstand the network's fast-paced and rapidly changing environment, we at CTDSec recommend that BurnItDAO team put in place a bug bounty program to encourage further and active analysis of the smart contract.

Copyright © ***CTDSec*** - All rights reserved

2.0 Coverage

2.1 Target Code and Revision

For this audit, we performed research, investigation, and review of the BurnItDAO contracts followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code files are considered in-scope for the review:

Source file:

firestarter-final.zip: SHA256 -

7c3eba6408b59a57fd308b88b3d5798e1adb46448e59c92e964ee9f323437bbd

Copyright © ***CTDSec*** - All rights reserved

2.2 Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

№	Issue description.	Checking status
1	Compiler warnings.	PASSED
2	Race conditions and Reentrancy. Cross-function race conditions.	PASSED
3	Possible delays in data delivery.	PASSED
4	Oracle calls.	PASSED
5	Front running.	PASSED
6	Timestamp dependence.	PASSED
7	Integer Overflow and Underflow.	PASSED
8	DoS with Revert.	PASSED
9	DoS with block gas limit.	PASSED
10	Methods execution permissions.	PASSED
11	Economy model. If application logic is based on an incorrect economic model, the application would not function correctly and participants would incur financial losses. This type of issue is most often found in bonus rewards systems, Staking and Farming contracts, Vault and Vesting contracts, etc.	PASSED
12	The impact of the exchange rate on the logic.	PASSED
13	Private user data leaks.	PASSED
14	Malicious Event log.	PASSED
15	Scoping and Declarations.	PASSED
16	Uninitialized storage pointers.	PASSED

Copyright © ***CTDSec*** - All rights reserved

17	Arithmetic accuracy.	PASSED
18	Design Logic.	PASSED
19	Cross-function race conditions.	PASSED
20	Safe Zeppelin module.	PASSED
21	Fallback function security.	PASSED
22	Overpowered functions / Owner privileges	PASSED

Copyright © ***CTDSec*** - All rights reserved

3.0 Security Issues

3.1 High severity issues

No high severity issues were found.

3.2 Medium severity issues

No medium severity issues were found.

3.3 Low severity issues [1 - Mitigated]

1. On-chain Randomness

Contract: BurnItDAO

Functions: reveal(), claim(), randomSeed(), randomN()

Issue: BurnItDAO uses random generated onchain.

Mitigation: Currently the contracts use PREVRANDAO that theoretically should mitigate this issue in ethereum network.

Copyright © ***CTDSec*** - All rights reserved

3.3 Informational Findings [1 - Not apply]

1. Reveal function doesn't check introduced bytes

Contract: BurnItDAO

Function: reveal()

Issue: Any secret bytes with length (4) allows to reveal their previously committed secret.

Fix: Verify if the _reveal Function Behaves as Intended [LibUser.sol]

Mitigation: Even if all players were using the same secret, the reduction in entropy would be minimal, as it is just an additional source and does not affect it.

Copyright © ***CTDSec*** - All rights reserved

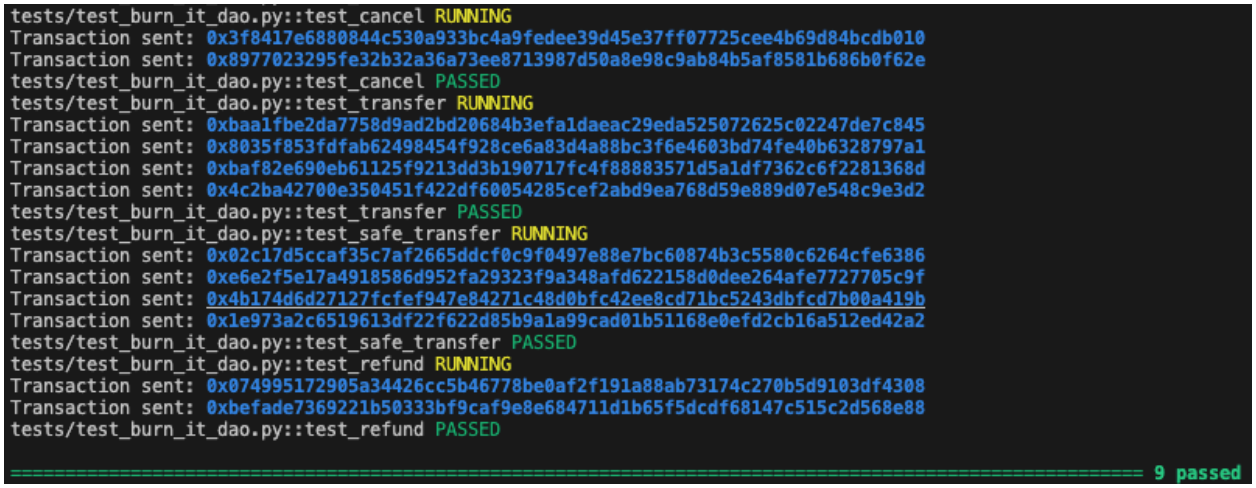
4.0 Testing coverage - python

During the testing phase, custom use cases were written to cover all the logic of contracts in python language. *Check “5 Annexes” to see the testing code.*

BurnitDAO contracts tests

```
tests/test_burn_it_dao.py::test_constructor RUNNING
Transaction sent: 0xfff2e9b346a4d338c8656dfde1465c2276102c98a2443734a6dcc5627141656d
Transaction sent: 0xdfdf9630e93851b17f9f0d33bbc360fecaff63eab3749138a98bdf629099ec19
Transaction sent: 0x6d7ac903af201264e2a22c4f10acf562af7ab8f619d0b2ded700cfaaa0c405d6
Transaction sent: 0x99085b09bd29030d81961a182d41cadb4675bf2c308d07ff1c7b465163c10542
Transaction sent: 0x0051e9685e9fc167bee9f4122802d04e767ecf05615a4bda0afc6b15891f93a4
Transaction sent: 0x0abd163ed6bc2ca0f0f7b3bdc7264dd6159b7853e24fc28eee6afdac4d8eb48e
Transaction sent: 0xecfa609aaabc22d9d130d012e871a99f7ef8d4884268828811052d971bdd329b
Transaction sent: 0x2935e9af1975fa1dbe46872b50b2719588a3e9846904850f1265779a9ced3a3c
Transaction sent: 0x04ed8b3254dfd454ad391f3cf43d98a8eab7bb0893f112417e4f13cf9d995197
tests/test_burn_it_dao.py::test_constructor PASSED
tests/test_burn_it_dao.py::test_mint RUNNING
Transaction sent: 0x5c975458e4a3a78f2aa7f7ddfea321de51314dc4062cfa2b5a04e1997fe31e7c
Transaction sent: 0x95947efa8b7aad62531ea9adcf5816f9997cd6ec222bcbb3477b03e4cbb8c5d2
Transaction sent: 0x650f91797de9c673d7d5132000924584714 90c827d9869eac6b3384bde3187f9
Transaction sent: 0x8dc79639d872c7dde1189154bf8dc99b5d7afbbbe558e39308aabbcb16026559
Transaction sent: 0x5af3833777098c1c747043a815b5cb2785cdda65d62b85a558795260f97f351b
tests/test_burn_it_dao.py::test_mint PASSED
tests/test_burn_it_dao.py::test_commit RUNNING
Transaction sent: 0x579159553907fbe6909c33f5d71c2647a20acbbc005df9f9f58e715244832bd6
Transaction sent: 0x92a5ce0d7ecce71fd437cc9ecbc318872cff746905b641fb8f946022557c260f
Transaction sent: 0x3d91d0c2e485fa609bf89a59e28733de3e600df3fc8ee4adf644e009a2a900d8
tests/test_burn_it_dao.py::test_commit PASSED
tests/test_burn_it_dao.py::test_reveal RUNNING
Transaction sent: 0xcc96e555078e3a46ddc15f34dfe96d9f36bd74f6c46a83d93be500b76f141a66
Transaction sent: 0x97effc7d2509d73788eb9f07d41d6ce8d97680fe561859dd7c31576b51a58620
Transaction sent: 0xb10afab8e6cef121343200b5e873b0d7877e876440501c6d13f2db1bf8e248f9
tests/test_burn_it_dao.py::test_reveal PASSED
tests/test_burn_it_dao.py::test_claim RUNNING
Transaction sent: 0xfd66af191ba537e9dffc35049f3904c6a441333caa4fc488451a1536ec8c2738
Transaction sent: 0x4ba7b079c4a0e0ba4ff1d13bd73cb0b1c34c1799c661f760fef59b57e3589c15
Transaction sent: 0x2e506ed04ab267d01ca711050c5a86be040bf830d0d3544f60ab4a044d7770b1
Transaction sent: 0xa81e6f340695e64591d75f8a0ac9ff8d65558de46235520c68c47de519112ec3
Transaction sent: 0xdba4bdd58b0f670b4bd8577a4ae576d50bb31e64a41f5e9dadb650db9c5185a2
tests/test_burn_it_dao.py::test_claim PASSED
```

Copyright © ***CTDSec*** - All rights reserved



Copyright © ***CTDSec*** - All rights reserved

5.0 Annexes

Testing code - burnitDAO:

```
from brownie import (
    reverts,
    BurnItDAO
)

from scripts.helpful_scripts import (
    ZERO_ADDRESS,
    DAY_TIMESTAMP,
    get_account,
    get_custom_error_hex,
    get_random_secret,
    evm_increase_time
)

from scripts.deploy import (
    deploy_burn_it_dao
)

def test_constructor(only_local):
    #arrange
    owner = get_account(0)
    team_addr = get_account(1)
    draw_addr = get_account(2)

    with reverts('typed error: ' +
get_custom_error_hex("ErrorTokenPrice()")):
        BurnItDAO.deploy('BurnItDAO', 'BID', 1, 0,
        30000, 10000, 1000, 10, team_addr, draw_addr, "token_uri",
        {"from": owner})
    with reverts('typed error: ' +
get_custom_error_hex("ErrorMaxTokens()")):
        BurnItDAO.deploy('BurnItDAO', 'BID', 1, 1e18 / 2,
        1, 10000, 1000, 10, team_addr, draw_addr, "token_uri",
        {"from": owner})
    with reverts('typed error: ' +
get_custom_error_hex("ErrorMaxTokens()")):
```

Copyright © ***CTDSec*** - All rights reserved

```
        BurnItDAO.deploy('BurnItDAO', 'BID', 1, 1e18 / 2,
        2**15, 10000, 1000, 10, team_addr, draw_addr, "token_uri",
        {"from": owner})
    with reverts('typed error: ' +
get_custom_error_hex("ErrorMaxWallet()")):
        BurnItDAO.deploy('BurnItDAO', 'BID', 1, 1e18 / 2,
        16000, 0, 1000, 10, team_addr, draw_addr, "token_uri",
        {"from": owner})
    with reverts('typed error: ' +
get_custom_error_hex("ErrorMaxAmount()")):
        BurnItDAO.deploy('BurnItDAO', 'BID', 1, 1e18 / 2,
        16000, 10000, 0, 10, team_addr, draw_addr, "token_uri",
        {"from": owner})
    with reverts('typed error: ' +
get_custom_error_hex("ErrorTeamSplit()")):
        BurnItDAO.deploy('BurnItDAO', 'BID', 1, 1e18 / 2,
        16000, 10000, 1000, 100, team_addr, draw_addr, "token_uri",
        {"from": owner})
    with reverts('typed error: ' +
get_custom_error_hex("ErrorTeamAddress()")):
        BurnItDAO.deploy('BurnItDAO', 'BID', 1, 1e18 / 2,
        16000, 10000, 1000, 10, ZERO_ADDRESS, draw_addr, "token_uri",
        {"from": owner})
    with reverts('typed error: ' +
get_custom_error_hex("ErrorDrawAddress()")):
        BurnItDAO.deploy('BurnItDAO', 'BID', 1, 1e18 / 2,
        16000, 10000, 1000, 10, team_addr, ZERO_ADDRESS, "token_uri",
        {"from": owner})
    with reverts('typed error: ' +
get_custom_error_hex("ErrorInvalidTokenURI()")):
        BurnItDAO.deploy('BurnItDAO', 'BID', 1, 1e18 / 2,
        16000, 10000, 1000, 10, team_addr, draw_addr, "token_uri",
        {"from": owner})

    contract = BurnItDAO.deploy('BurnItDAO', 'BID', 1, 1e18 / 2,
        16000, 10000, 1000, 10, team_addr, draw_addr,
"some_long_token_uri",
        {"from": owner})

def test_mint(only_local):
    #arrange
    owner = get_account(0)
```

```
    team_addr = get_account(1)
    draw_addr = get_account(2)
    other = get_account(3)

    token_price = 1e18 / 2
    max_tokens = 12
    max_wallet = 10
    max_amount = 5
    burn_it_dao = BurnItDAO.deploy(
        'BurnItDAO', 'BID', 1, token_price, max_tokens, max_wallet,
max_amount, 10,
        team_addr, draw_addr, "some_long_token_uri", {"from": owner})

    # assert
    with reverts('typed error: ' +
get_custom_error_hex("ErrorMintTxAmount()")):
        burn_it_dao.mint(1000, {"from": owner})
    with reverts('typed error: ' +
get_custom_error_hex("ErrorMintTxPrice()")):
        burn_it_dao.mint(5, {"from": owner})

    # Mint 10
    burn_it_dao.mint(5, {"from": owner, "value": 5 * token_price})
    burn_it_dao.mint(5, {"from": owner, "value": 5 * token_price})

    with reverts('typed error: ' +
get_custom_error_hex("ErrorMintMaxWallet()")):
        burn_it_dao.mint(1, {"from": owner, "value": 1 * token_price})

    with reverts('typed error: ' +
get_custom_error_hex("ErrorMintMaxTokens()")):
        burn_it_dao.mint(3, {"from": other, "value": 3 * token_price})

    burn_it_dao.mint(2, {"from": other, "value": 2 * token_price})
    with reverts('typed error: ' +
get_custom_error_hex("ErrorMintNotActive()")):
        burn_it_dao.mint(1, {"from": other, "value": 1 * token_price})

def test_commit(only_local):
    #arrange
    owner = get_account(0)
    team_addr = get_account(1)
```

Copyright © ***CTDSec*** - All rights reserved

```
    draw_addr = get_account(2)
    other = get_account(3)

    token_price = 1e18 / 2
    max_tokens = 12
    max_wallet = 10
    max_amount = 5
    burn_it_dao = BurnItDAO.deploy(
        'BurnItDAO', 'BID', 1, token_price, max_tokens, max_wallet,
max_amount, 10,
        team_addr, draw_addr, "some_long_token_uri", {"from": owner})

    #assert
    random_bytes = b'ramdom_bytes_string'
    # claim without tokens
    with reverts('typed error: ' +
get_custom_error_hex("ErrorClaimInvalidUser()")):
        burn_it_dao.commit(random_bytes, {"from": owner})

    burn_it_dao.mint(5, {"from": owner, "value": 5 * token_price})
    burn_it_dao.mint(5, {"from": owner, "value": 5 * token_price})
    # claim game not running
    with reverts('typed error: ' +
get_custom_error_hex("ErrorGameNotRunning()")):
        burn_it_dao.commit(random_bytes, {"from": owner})
    burn_it_dao.mint(2, {"from": other, "value": 2 * token_price})

    burn_it_dao.commit(random_bytes, {"from": other})

    # commit again
    with reverts('typed error: ' +
get_custom_error_hex("ErrorCommitPrevious()")):
        burn_it_dao.commit(random_bytes, {"from": other})

def test_reveal(only_local):
    #arrange
    owner = get_account(0)
    team_addr = get_account(1)
    draw_addr = get_account(2)
    other = get_account(3)

    token_price = 1e18 / 2
```

Copyright © ***CTDSec*** - All rights reserved

```
    max_tokens = 12
    max_wallet = 10
    max_amount = 5
    burn_it_dao = BurnItDAO.deploy(
        'BurnItDAO', 'BID', 1, token_price, max_tokens, max_wallet,
max_amount, 10,
        team_addr, draw_addr, "some_long_token_uri", {"from": owner})

    #assert
    random_hash_secret = get_random_secret()
    # start game
    burn_it_dao.mint(5, {"from": other, "value": 5 * token_price})
    burn_it_dao.mint(5, {"from": other, "value": 5 * token_price})
    burn_it_dao.mint(2, {"from": owner, "value": 2 * token_price})
    # commit something
    burn_it_dao.commit(random_hash_secret, {"from": other})

    with reverts('typed error: ' +
get_custom_error_hex("ErrorRevealLength()")):
        burn_it_dao.reveal(b'not_commit', {"from": other})

    secret = 0x80000000
    with reverts('typed error: ' +
get_custom_error_hex("ErrorRevealDenied()")):
        burn_it_dao.reveal(secret, {"from": owner})

    tx = burn_it_dao.reveal(secret, {"from": other})
    assert tx.events["Reveal"] is not None
    assert tx.events["Reveal"][0]['from'] == other

    # reveal the secret again
    with reverts('typed error: ' +
get_custom_error_hex("ErrorRevealDenied()")):
        burn_it_dao.reveal(secret, {"from": other})

def test_claim(only_local):
    #arrange
    owner = get_account(0)
    team_addr = get_account(1)
    draw_addr = get_account(2)
    other = get_account(3)
```

Copyright © ***CTDSec*** - All rights reserved

```
    token_price = 1e18 / 2
    max_tokens = 12
    max_wallet = 10
    max_amount = 5
    burn_it_dao = BurnItDAO.deploy(
        'BurnItDAO', 'BID', 1, token_price, max_tokens, max_wallet,
max_amount, 10,
        team_addr, draw_addr, "some_long_token_uri", {"from": owner})

    #assert
    with reverts('typed error: ' +
get_custom_error_hex("OwnerQueryForNonexistentToken()")):
        burn_it_dao.claim(99, {"from": other})
    burn_it_dao.mint(1, {"from": other, "value": 1 * token_price})
    with reverts('typed error: ' +
get_custom_error_hex("ErrorClaimInvalidOwner()")):
        burn_it_dao.claim(0, {"from": owner})
    with reverts('typed error: ' +
get_custom_error_hex("ErrorClaimUnavailable()")):
        burn_it_dao.claim(0, {"from": other})
    # Mint some more
    burn_it_dao.mint(4, {"from": other, "value": 4 * token_price})
    burn_it_dao.mint(5, {"from": other, "value": 5 * token_price})
    burn_it_dao.mint(2, {"from": owner, "value": 2 * token_price})

    random_hash_secret = get_random_secret()
    burn_it_dao.commit(random_hash_secret, {"from": other})
    tx = burn_it_dao.claim(0, {"from": other})
    #print(tx.events)
    with reverts('typed error: ' +
get_custom_error_hex("ErrorClaimPermissionDenied()")):
        burn_it_dao.claim(10, {"from": owner})

    burn_it_dao.commit(random_hash_secret, {"from": owner})
    tx = burn_it_dao.claim(10, {"from": owner})
    #print(tx.events)

    # ErrorClaimInvalidToken
    with reverts():
        burn_it_dao.claim(0, {"from": other})

def test_cancel(only_local):
```

```
    #arrange
    owner = get_account(0)
    team_addr = get_account(1)
    draw_addr = get_account(2)
    other = get_account(3)

    token_price = 1e18 / 2
    max_tokens = 12
    max_wallet = 10
    max_amount = 5
    burn_it_dao = BurnItDAO.deploy(
        'BurnItDAO', 'BID', 1, token_price, max_tokens, max_wallet,
max_amount, 10,
        team_addr, draw_addr, "some_long_token_uri", {"from": owner})

    #assert
    burn_it_dao.mint(5, {"from": other, "value": 5 * token_price})
    burn_it_dao.mint(5, {"from": other, "value": 5 * token_price})
    burn_it_dao.mint(2, {"from": owner, "value": 2 * token_price})
    # No timelock completed
    with reverts():
        burn_it_dao.cancel({"from": other})
    evm_increase_time(DAY_TIMESTAMP * 90)
    burn_it_dao.setBaseURI("random_url_to_increse_block", {"from": owner})
    burn_it_dao.setBaseURI("random_url_to_increse_block", {"from": owner})

    with reverts('typed error: ' +
get_custom_error_hex("ErrorMintNotActive()")):
        burn_it_dao.cancel({"from": other})

    # can cancel game
    burn_it_dao = BurnItDAO.deploy(
    'BurnItDAO', 'BID', 1, token_price, max_tokens, max_wallet, max_amount,
10,
    team_addr, draw_addr, "some_long_token_uri", {"from": owner})
    burn_it_dao.mint(5, {"from": other, "value": 5 * token_price})
    evm_increase_time(DAY_TIMESTAMP * 90)
    burn_it_dao.setBaseURI("random_url_to_increse_block", {"from": owner})
    burn_it_dao.setBaseURI("random_url_to_increse_block", {"from": owner})
    tx = burn_it_dao.cancel({"from": other})
    assert tx.events["GameCancelled"] is not None
```

Copyright © ***CTDSec*** - All rights reserved

```
def test_transfer(only_local):
    #arrange
    owner = get_account(0)
    team_addr = get_account(1)
    draw_addr = get_account(2)
    other = get_account(3)
    extra = get_account(4)

    token_price = 1e18 / 2
    burn_it_dao = deploy_burn_it_dao(owner, team_addr, draw_addr)
    #assert
    with reverts('typed error: ' +
get_custom_error_hex("OperatorQueryForNonexistentToken()")):
        burn_it_dao.transferFrom(owner, other, 1, {"from": other})

    burn_it_dao.mint(1, {"from": owner, "value": 1 * token_price})

    with reverts('typed error: ' +
get_custom_error_hex("TransferFromIncorrectOwner()")):
        burn_it_dao.transferFrom(owner, other, 0, {"from": other})
    with reverts('typed error: ' +
get_custom_error_hex("TransferFromIncorrectOwner()")):
        burn_it_dao.transferFrom(owner, owner, 0, {"from": owner})

    assert burn_it_dao.isTokenOwner(owner, 0) == True
    tx = burn_it_dao.transferFrom(owner, other, 0, {"from": owner})
    assert burn_it_dao.isTokenOwner(owner, 0) == False
    burn_it_dao.mint(5, {"from": other, "value": 5 * token_price})
    burn_it_dao.mint(4, {"from": other, "value": 4 * token_price})
    burn_it_dao.mint(2, {"from": owner, "value": 2 * token_price})

    random_hash_secret = get_random_secret()
    burn_it_dao.commit(random_hash_secret, {"from": other})

    with reverts('typed error: ' +
get_custom_error_hex("ErrorTransferDenied()")):
        burn_it_dao.transferFrom(other, extra, 0, {"from": other})

def test_safe_transfer(only_local):
    #arrange
    owner = get_account(0)
    team_addr = get_account(1)
```

Copyright © ***CTDSec*** - All rights reserved

```
    draw_addr = get_account(2)
    other = get_account(3)
    extra = get_account(4)

    token_price = 1e18 / 2
    burn_it_dao = deploy_burn_it_dao(owner, team_addr, draw_addr)
    random_data = get_random_secret()
    #assert
    with reverts('typed error: ' +
get_custom_error_hex("OperatorQueryForNonexistentToken()")):
        burn_it_dao.safeTransferFrom(owner, other, 1, random_data, {"from":
other})

    burn_it_dao.mint(1, {"from": owner, "value": 1 * token_price})

    with reverts('typed error: ' +
get_custom_error_hex("TransferFromIncorrectOwner()")):
        burn_it_dao.safeTransferFrom(owner, other, 0, random_data, {"from":
other})
    with reverts('typed error: ' +
get_custom_error_hex("TransferFromIncorrectOwner()")):
        burn_it_dao.safeTransferFrom(owner, owner, 0, random_data, {"from":
owner})

    assert burn_it_dao.isTokenOwner(owner, 0) == True
    tx = burn_it_dao.safeTransferFrom(owner, other, 0, {"from": owner})
    assert burn_it_dao.isTokenOwner(owner, 0) == False
    burn_it_dao.mint(5, {"from": other, "value": 5 * token_price})
    burn_it_dao.mint(4, {"from": other, "value": 4 * token_price})
    burn_it_dao.mint(2, {"from": owner, "value": 2 * token_price})

    random_hash_secret = get_random_secret()
    burn_it_dao.commit(random_hash_secret, {"from": other})

    with reverts('typed error: ' +
get_custom_error_hex("ErrorTransferDenied()")):
        burn_it_dao.safeTransferFrom(other, extra, 0, random_data, {"from":
other})

def test_refund(only_local):
    #arrange
    owner = get_account(0)
```

Copyright © ***CTDSec*** - All rights reserved

```
    team_addr = get_account(1)
    draw_addr = get_account(2)
    other = get_account(3)
    extra = get_account(4)

    token_price = 500000000000000000
    burn_it_dao = deploy_burn_it_dao(owner, team_addr, draw_addr)
    #assert

    # cancel a game first
    burn_it_dao.mint(5, {"from": other, "value": 5 * token_price})
    evm_increase_time(DAY_TIMESTAMP * 90)
    burn_it_dao.setBaseURI("random_url_to_increse_block", {"from": owner})
    burn_it_dao.setBaseURI("random_url_to_increse_block", {"from": owner})
    burn_it_dao.cancel({"from": other})


    with reverts('typed error: ' +
get_custom_error_hex("ErrorMintNonRefundable()")):
        burn_it_dao.refund(2, other, {"from": other})

    with reverts('typed error: ' +
get_custom_error_hex("ErrorNonRefundable()")):
        burn_it_dao.refund(1, owner, {"from": other})

    tx = burn_it_dao.refund(1, other, {"from": other})
    assert tx.events["Deposited"] is not None
    assert tx.events["Deposited"][0]['payee'] == other

    # Another refund with two addrs
    burn_it_dao = deploy_burn_it_dao(owner, team_addr, draw_addr)
    burn_it_dao.mint(3, {"from": other, "value": 3 * token_price})
    burn_it_dao.mint(2, {"from": extra, "value": 2 * token_price})
    evm_increase_time(DAY_TIMESTAMP * 90)
    burn_it_dao.setBaseURI("random_url_to_increse_block", {"from": owner})
    burn_it_dao.setBaseURI("random_url_to_increse_block", {"from": owner})
    burn_it_dao.cancel({"from": extra})

    tx = burn_it_dao.refund(1, other, {"from": other})
    assert tx.events["Deposited"] is not None
    assert tx.events["Deposited"][0]['payee'] == other
```

Copyright © ***CTDSec*** - All rights reserved

```
    tx = burn_it_dao.refund(1, extra, {"from": extra})
    assert tx.events["Deposited"] is not None
    assert tx.events["Deposited"][0]['payee'] == extra
```

Copyright © ***CTDSec*** - All rights reserved

6.0 Summary of the audit

The contract does not have high or medium vulnerabilities, the code follows good development practices and has very clean code.

Copyright © ***CTDSec*** - All rights reserved