hexens × Bahamut

Apr.23

SECURITY REVIEW REPORT FOR BAHAMUT

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

AUDIT LED BY

VAHE KARAPETYAN

Co-founder / CTO | Hexens

Audit Starting Date
06.04.2023

Audit Completion Date
01.05.2023

---

hexens × Bahamut

METHODOLOGY

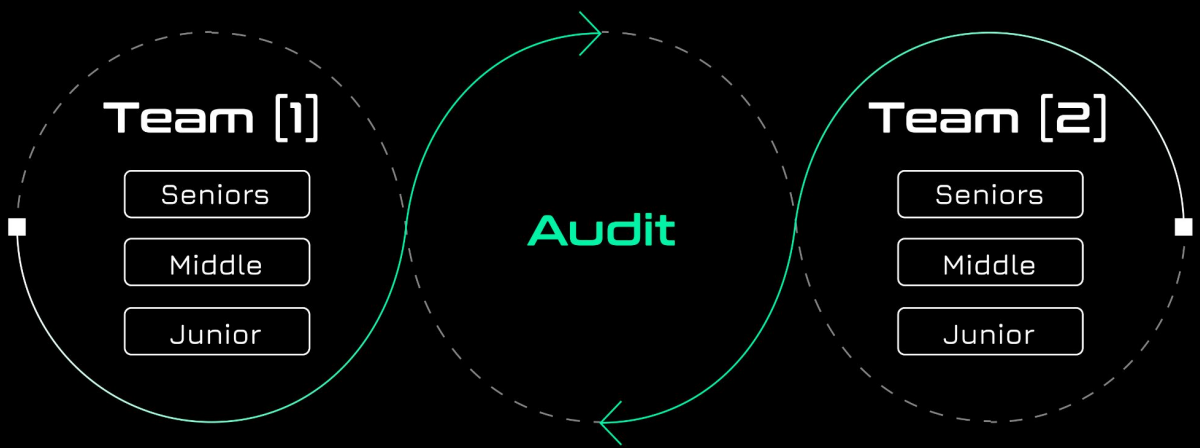
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor* Audit

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

Bahamut blockchain

This audit covered the new L1 blockchain of Bahamut, which is a fork of the Ethereum blockchain with a modified Proof of Stake and Activity (POSA) consensus mechanism.

Our security assessment was a full review of both the execution layer and the consensus layer.

During our audit, we have identified 1 critical severity vulnerability in the part of the code that is responsible for calculating activity points for memory usage. This can lead to a situation where a validator can receive more activity points than intended by the activity mechanism.

We have also identified 2 high severity vulnerabilities, various minor vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

Deposit.sol

The purpose of this audit was to verify the security and identify possible vulnerabilities in the deposit.sol smart contract.

Based on the results of the conducted audit, we are pleased to report that no vulnerabilities were found in the deposit.sol smart contract.

SCOPE

The analyzed resources are located on:
https://github.com/fastexlabs/bahamut-consensus/commit/2c84b03bdfc4b2fe91cfaaa5b5d3524b750a2507
https://github.com/fastexlabs/bahamut-execution/commit/af75d5f6c6ab5a33f6a1ac86c5c443e7be943cf1

The issues described in this report were fixed.

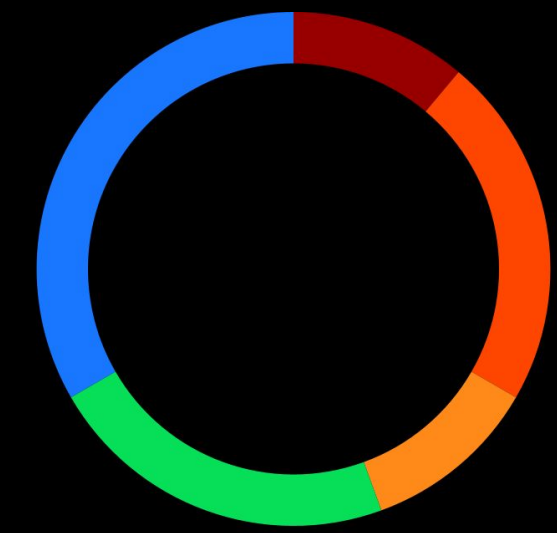
SUMMARY

Bahamut Blockchain Certificate

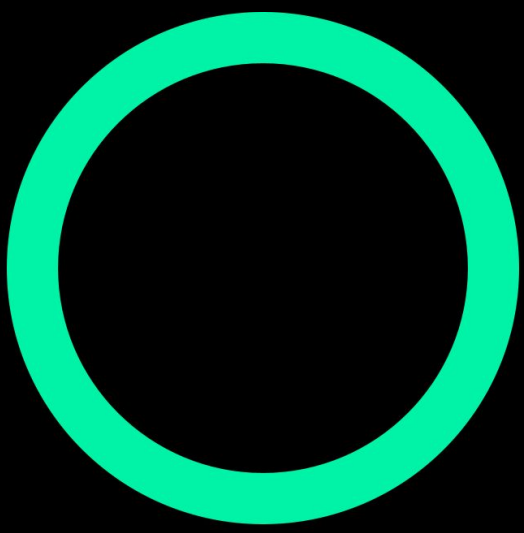
SEVERITY	NUMBER OF FINDINGS
CRITICAL	1
HIGH	2
MEDIUM	1
LOW	2
INFORMATIONAL	3

TOTAL: 9

SEVERITY



STATUS



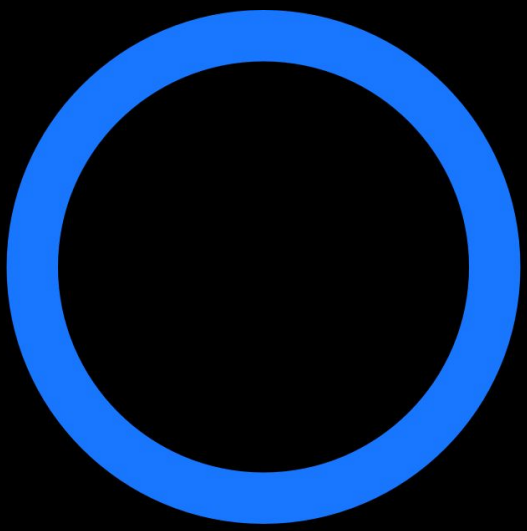
SUMMARY

Deposit.sol Certificate

SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	0
MEDIUM	0
LOW	0
INFORMATIONAL	0

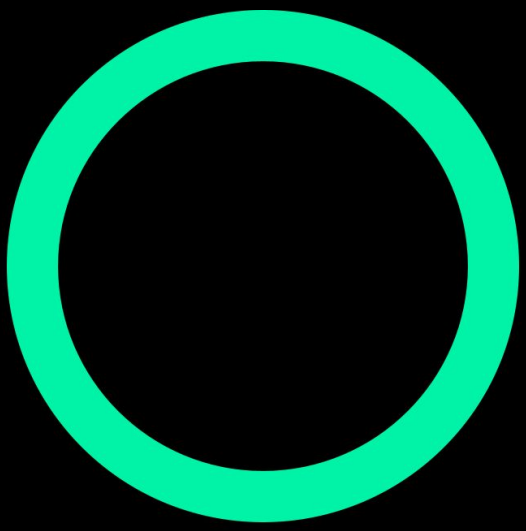
TOTAL: 0

SEVERITY



● No issues

STATUS



● No issues

WEAKNESSES

This section contains the list of discovered weaknesses.

FTN-4. ACTIVITY SCORES ABUSE

SEVERITY: Critical

PATH: bahamut-execution-master/core/vm/evm.go/memoryGas()

REMEDIATION: we suggest not counting gas either when creating an EOA-to-contract transaction or when creating a contract-to-contract transaction

STATUS: fixed

DESCRIPTION:

calldata is counted in two different ways: when sending a transaction from an EOA to a contract, and when sending a transaction from a contract to a contract. Function **memoryGas()** does not take this fact into account. When sending a transaction from contract-to-contract, gas is calculated as follows - for zero bytes, the price for gas is 4, for non-zero bytes, the price for gas is 16.

When sending a transaction from a contract-to-contract, function **memoryGas()** will calculate the gas according to the standard formula, while the user will receive more activity points than he spent gas.

For example, by sending such a transaction, and filling the block with 30 million gas, we will receive 100 million gas.

```
func (evm *EVM) memoryGas(data []byte) (uint64, error) {
  var gas uint64
  rules := evm.ChainConfig().Rules(evm.Context.BlockNumber, evm.Context.Random != nil)
  if len(data) > 0 {
    // Zero and non-zero bytes are priced differently
    var nz uint64
    for _, byt := range data {
      if byt != 0 {
        nz++
      }
    }
    // Make sure we don't exceed uint64 for all data combinations
    nonZeroGas := params.TxDataNonZeroGasFrontier
    if rules.IsIstanbul {
      nonZeroGas = params.TxDataNonZeroGasEIP2028
    }
    if (math.MaxUint64-gas)/nonZeroGas < nz {
      return 0, ErrGasUintOverflow
    }
    gas += nz * nonZeroGas

    z := uint64(len(data)) - nz
    if (math.MaxUint64-gas)/params.TxDataZeroGas < z {
      return 0, ErrGasUintOverflow
    }
    gas += z * params.TxDataZeroGas
  }

  return gas, nil
}
```

FTN-1. WRONG ACTIVITY CALCULATIONS

SEVERITY: High

PATH: bahamut-execution-master/core/vm/evm.go

REMEDIATION: remove activity score calculations from DelegateCall and CallCode

STATUS: fixed

DESCRIPTION:

In functions **CallCode** and **DelegateCall**, activity parameters are incorrectly calculated.

The problem is that when the **DelegateCall** and **CallCode** are called, the execution context changes. This can cause problems when counting activity scores.

For example, when using proxy pattern, activity will be credited to the implementation contract. This can cause problems for the validator, since when the implementation changes, all activity points will be lost.

CallCode:

```
func (evm *EVM) CallCode(caller ContractRef, addr common.Address, input []byte, gas uint64, value *big.Int) (ret []byte,
leftOverGas uint64, err error) {
  // Fail if we're trying to execute above the call depth limit
  if evm.depth > int(params.CallCreateDepth) {
    return nil, gas, ErrDepth
  }

  initialGas := gas
  // Fail if we're trying to transfer more than the available balance
  // Note although it's noop to transfer X ether to caller itself. But
  // if caller doesn't have enough balance, it would be an error to allow
  // over-charging itself. So the check here is necessary.
  if !evm.Context.CanTransfer(evm.StateDB, caller.Address(), value) {
    return nil, gas, ErrInsufficientBalance
  }
  snapshot := evm.StateDB.Snapshot()

  // Invoke tracer hooks that signal entering/exiting a call frame
  if evm.Config.Debug {
    evm.Config.Tracer.CaptureEnter(CALLCODE, caller.Address(), addr, input, gas, value)
    defer func(startGas uint64) {
      evm.Config.Tracer.CaptureExit(ret, startGas-gas, err)
    }(gas)
  }

  // It is allowed to call precompiles, even via delegatecall
  if p, isPrecompile := evm.precompile(addr); isPrecompile {
    ret, gas, err = RunPrecompiledContract(p, input, gas)
  } else {
    addrCopy := addr
    // Initialise a new contract and set the code that is to be used by the EVM.
    // The contract is a scoped environment for this execution context only.
    contract := NewContract(caller, AccountRef(caller.Address()), value, gas)
    contract.SetCallCode(&addrCopy, evm.StateDB.GetCodeHash(addrCopy), evm.StateDB.GetCode(addrCopy))
    ret, err = evm.interpreter.Run(contract, input, false)
    gas = contract.Gas
```

```
memGas, err := evm.memoryGas(input)
if err != nil {
  return nil, gas, err
}
if caller.Address() != evm.Origin {
  memGas = 0
}
evm.StateDB.AddActivity(addrCopy, initialGas-contract.Gas-contract.OthersGas+memGas)
evm.StateDB.AddActivities(&types.Activity{
  Address:      addrCopy,
  Activity:     evm.StateDB.GetActivity(addrCopy),
  DeltaActivity: initialGas - contract.Gas - contract.OthersGas,
})
}
if err != nil {
evm.StateDB.RevertToSnapshot(snapshot)
if err != ErrExecutionReverted {
  gas = 0
}
}
return ret, gas, err
}
```

DelegateCall:

```
func (evm *EVM) DelegateCall(caller ContractRef, addr common.Address, input []byte, gas uint64) (ret []byte,
leftOverGas uint64, err error) {
  // Fail if we're trying to execute above the call depth limit
  if evm.depth > int(params.CallCreateDepth) {
    return nil, gas, ErrDepth
  }

  initialGas := gas

  snapshot := evm.StateDB.Snapshot()

  // Invoke tracer hooks that signal entering/exiting a call frame
  if evm.Config.Debug {
    evm.Config.Tracer.CaptureEnter(DELEGATECALL, caller.Address(), addr, input, gas, nil)
    defer func(startGas uint64) {
      evm.Config.Tracer.CaptureExit(ret, startGas-gas, err)
    }(gas)
  }

  // It is allowed to call precompiles, even via delegatecall
  if p, isPrecompile := evm.precompile(addr); isPrecompile {
    ret, gas, err = RunPrecompiledContract(p, input, gas)
  } else {
    addrCopy := addr
    // Initialise a new contract and make initialise the delegate values
    contract := NewContract(caller, AccountRef(caller.Address()), nil, gas).AsDelegate()
    contract.SetCallCode(&addrCopy, evm.StateDB.GetCodeHash(addrCopy), evm.StateDB.GetCode(addrCopy))
    ret, err = evm.interpreter.Run(contract, input, false)
    gas = contract.Gas

    evm.StateDB.AddActivity(addrCopy, initialGas-contract.Gas-contract.OthersGas)
    evm.StateDB.AddActivities(&types.Activity{
      Address:      addrCopy,
      Activity:     evm.StateDB.GetActivity(addrCopy),
      DeltaActivity: initialGas - contract.Gas - contract.OthersGas,
    })
  }
```

```
if err != nil {
  evm.StateDB.RevertToSnapshot(snapshot)
  if err != ErrExecutionReverted {
    gas = 0
  }
}
return ret, gas, err
}
```

FTN-2. VALIDATOR CAN ADD MORE THAN ONE CONTRACT

SEVERITY: High

PATH:
bahamut-consensus-master/beacon-chain/core/altair/deposit.go/ProcessDeposit(),
bahamut-consensus-master/beacon-chain/core/helpers/contracts.go/appendValidatorContractsWithVal()

REMEDIATION: delete the old contract before adding a new one

STATUS: fixed

DESCRIPTION:

Function **ProcessDeposit()** is responsible for adding new validators and their contracts. This function calls function **appendValidatorContractsWithVal()** which adds a new contract to the given validator. But each validator can only have one contract tied to its address.

ProcessDeposit():

```
func ProcessDeposit(beaconState state.BeaconState, deposit *ethpb.Deposit, verifySignature bool)
(state.BeaconState, bool, error) {
  var newValidator bool
  if err := verifyDeposit(beaconState, deposit); err != nil {
    if deposit == nil || deposit.Data == nil {
      return nil, newValidator, err
    }
    return nil, newValidator, errors.Wrapf(err, "could not verify deposit from %#x",
bytesutil.Trunc(deposit.Data.PublicKey))
  }
  if err := beaconState.SetEth1DepositIndex(beaconState.Eth1DepositIndex() + 1); err != nil {
    return nil, newValidator, err
  }
  pubKey := deposit.Data.PublicKey
  amount := deposit.Data.Amount
  contract := deposit.Data.DeployedContract
  owner, contractExist := beaconState.ValidatorIndexByContractAddress(bytesutil.ToBytes20(contract))
  if contractExist {
    log.Debugf("Contract %x already regestered by validator %d. Register new validator with 0x0 contract address",
contract, owner)
  }

  index, ok := beaconState.ValidatorIndexByPubkey(bytesutil.ToBytes48(pubKey))
  if !ok {
    if verifySignature {
      domain, err := signing.ComputeDomain(params.BeaconConfig().DomainDeposit, nil, nil)
      if err != nil {
        return nil, newValidator, err
      }
      if err := verifyDepositDataSigningRoot(deposit.Data, domain); err != nil {
        // Ignore this error as in the spec pseudo code.
        log.WithError(err).Debug("Skipping deposit: could not verify deposit data signature")
        return beaconState, newValidator, nil
      }
    }

    effectiveBalance := amount - (amount % params.BeaconConfig().EffectiveBalanceIncrement)
    if params.BeaconConfig().MaxEffectiveBalance < effectiveBalance {
      effectiveBalance = params.BeaconConfig().MaxEffectiveBalance
    }
```

```
if err := beaconState.AppendValidator(&ethpb.Validator{
  PublicKey:                  pubKey,
  WithdrawalCredentials:      deposit.Data.WithdrawalCredentials,
  ActivationEligibilityEpoch: params.BeaconConfig().FarFutureEpoch,
  ActivationEpoch:            params.BeaconConfig().FarFutureEpoch,
  ExitEpoch:                  params.BeaconConfig().FarFutureEpoch,
  WithdrawableEpoch:          params.BeaconConfig().FarFutureEpoch,
  EffectiveBalance:           effectiveBalance,
}); err != nil {
  return nil, newValidator, err
}
newValidator = true
if err := beaconState.AppendBalance(amount); err != nil {
  return nil, newValidator, err
}
if err := beaconState.AppendActivity(0); err != nil {
  return nil, newValidator, err
}
contracts := [][]byte{contract}
if contractExist {
  contracts = [][]byte{make([]byte, 20)}
}
if err := beaconState.AppendContracts(&ethpb.ContractsContainer{
  Contracts: contracts,
}); err != nil {
  return nil, newValidator, err
}
} else {
if err := helpers.IncreaseBalance(beaconState, index, amount); err != nil {
  return nil, newValidator, err
}
if !contractExist && !isZeroContract(contract) {
  if err := helpers.AppendValidatorContracts(beaconState, index, contract); err != nil {
    return nil, newValidator, err
  }
}
}

return beaconState, newValidator, nil
}
```

appendValidatorContractsWithVal()

```
func appendValidatorContractsWithVal(cc *ethpb.ContractsContainer, contract []byte)
*ethpb.ContractsContainer {
  contracts := cc.Contracts
  contracts = append(contracts, contract)
  cc.Contracts = contracts
  return cc
}
```

FTN-3. INCORRECTLY ADDING A VALIDATOR CONTRACT

SEVERITY: Medium

PATH:
bahamut-consensus-master/beacon-chain/core/altair/deposit.go/ProcessDeposit()

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

When adding a validator's contract, if the contract exists, a new contract object with a zero address is added.

The addition of a null contract is due to the fact that the check for the existence of a contract and its addition occur in different **if** blocks.

```
func ProcessDeposit(beaconState state.BeaconState, deposit *ethpb.Deposit, verifySignature bool)
(state.BeaconState, bool, error) {
  [...]
  index, ok := beaconState.ValidatorIndexByPubkey(bytesutil.ToBytes48(pubKey))
  if !ok {
    [...]
    if contractExist {
      contracts = [][]byte{make([]byte, 20)}
    }
    if err := beaconState.AppendContracts(&ethpb.ContractsContainer{
      Contracts: contracts,
    }); err != nil {
      return nil, newValidator, err
    }
  } else {
    [...]
  }

  return beaconState, newValidator, nil
}
```

For example:

```
func ProcessDeposit(beaconState state.BeaconState, deposit *ethpb.Deposit, verifySignature bool)
(state.BeaconState, bool, error) {
  [...]
  index, ok := beaconState.ValidatorIndexByPubkey(bytesutil.ToBytes48(pubKey))
  if !ok {
     [...]
     if contractExist {
        contracts = [][]byte{make([]byte, 20)}
     } else {
        err := beaconState.AppendContracts(&ethpb.ContractsContainer{Contracts: contracts,})
        if err != nil {
           return nil, newValidator, err
        }
     }
  } else {
     [...]
  }

  return beaconState, newValidator, nil
}
```

FTN-6. HARDCODED VALUE INSTEAD OF CONSTANT VARIABLE

SEVERITY: Low

PATH:
bahamut-consensus-master/beacon-chain/core/epoch/epoch_processing.go:ProcessTransactionsGasPerPeriodUpdate()

REMEDIATION: use constant variables instead of hardcoded values

STATUS: fixed

DESCRIPTION:

Function **ProcessTransactionsGasPerPeriodUpdate()** calculates the gas update transaction per period. In the calculation of gas for the period, it is multiplied by the constant cost of the transaction. This value is hardcoded, which can lead to problems when changing the constant cost per transaction.

```
func ProcessTransactionsGasPerPeriodUpdate(state state.BeaconState) (state.BeaconState, error) {
  period := uint64(params.BeaconConfig().EpochsPerActivityPeriod)
  txsPerPeriod := ((state.TransactionsGasPerPeriod()+state.TransactionsPerLatestEpoch()*21000)*period -
state.TransactionsGasPerPeriod()) / periodc
  if err := state.SetTransactionsGasPerPeriod(txsPerPeriod); err != nil {
    return nil, err
  }
  if err := state.SetTransactionsPerLatestEpoch(0); err != nil {
    return nil, err
  }

  return state, nil
}
```

FTN-9. CODE OPTIMISATION

SEVERITY: Low

PATH:
bahamut-execution-master/core/state_transition.go:TransitionDb()

REMEDIATION: we would suggest implementing the logging logic, in which only those activity parameters that were changed during the transaction will be included in the logs

STATUS: fixed

DESCRIPTION:

Function **TransitionDb()** will transition the state by applying the current message and returning the evm execution result. During the execution of the **TransitionDb()**, activity scores are logged before and after refunds. Since function **TransitionDb()** will be called during the execution of any transaction, logging will also occur during any transaction, which can significantly slow down the blockchain as a whole.

```
func (st *StateTransition) TransitionDb() (*ExecutionResult, error) {
  [...]

  for _, act := range st.state.GetCurrentActivities() {
    log.Debug("Activities before refund", "address", act.Address, "activity", act.Activity, "delta",
act.DeltaActivity) // @note logs in for loops
  }

  [...]

  for _, act := range st.state.GetCurrentActivities() {
    log.Debug("Activities after refund", "address", act.Address, "activity", act.Activity, "delta",
act.DeltaActivity)
  }

  [...]
}
```

FTN-5. REDUNDANT CHECK

SEVERITY: Informational

PATH: bahamut-execution-master/core/vm/evm.go:CallCode()

REMEDIATION: remove redundant checks to optimize the code

STATUS:

DESCRIPTION: fixed

At runtime, function **CallCode()** checks if the address of the caller is the address that created the transaction. But due to the way opcode **CallCode()** works, the address of caller never changes.

```
func (evm *EVM) CallCode(caller ContractRef, addr common.Address, input []byte, gas uint64, value
*big.Int) (ret []byte, leftOverGas uint64, err error) {
  [...]
  if p, isPrecompile := evm.precompile(addr); isPrecompile {
    ret, gas, err = RunPrecompiledContract(p, input, gas)
  } else {
    [...]
    if caller.Address() != evm.Origin {
      memGas = 0
    }
    [...]
  }
  [...]
}
```

FTN-7. INCONSISTENCY BETWEEN DOCUMENTATION AND FUNCTION

SEVERITY: Informational

PATH:
bahamut-consensus-master/beacon-chain/core/epoch/epoch_processing.go:ProcessNonStakersGasPerPeriodUpdate()

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The documentation for function **ProcessNonStakersGasPerPeriodUpdate()** uses the name of function **ProcessTransactionsGasPerPeriodUpdate()**.

```
// ProcessTransactionsGasPerPeriodUpdate processes the updates to non stakers gas per period.
func ProcessNonStakersGasPerPeriodUpdate(state state.BeaconState) (state.BeaconState, error) {
  period := uint64(params.BeaconConfig().EpochsPerActivityPeriod)
  nonStakersGasPerPeriod := ((state.NonStakersGasPerPeriod()+state.NonStakersGasPerEpoch())*period -
state.NonStakersGasPerPeriod()) / period
  if err := state.SetNonStakersGasPerPeriod(nonStakersGasPerPeriod); err != nil {
    return nil, err
  }
  if err := state.SetNonStakersGasPerEpoch(0); err != nil {
    return nil, err
  }

  return state, nil
}
```

FTN-8. REDUNDANT CHECK

SEVERITY: Informational

PATH: bahamut-execution-master/core/vm/evm.go:memoryGas()

REMEDIATION: remove redundant checks to optimize code performance

STATUS: fixed

DESCRIPTION:

Function **memoryGas()** calculates the price of **calldata** during the execution of the **Call()** function. During the execution of the function **memoryGas()**, checks are made to prevent variable **gas** from overflowing. These checks are redundant since the **blocksize** will not allow variable **gas** to overflow.

```
func (evm *EVM) memoryGas(data []byte) (uint64, error) {
  [...]
  if len(data) > 0 {
    [...]
    nonZeroGas := params.TxDataNonZeroGasFrontier
    if rules.IsIstanbul {
      nonZeroGas = params.TxDataNonZeroGasEIP2028
    }
    if (math.MaxUint64)/nonZeroGas < nz { // @note redundant check (block limit)
      return 0, ErrGasUintOverflow
    }
    gas += nz * nonZeroGas

    z := uint64(len(data)) - nz
    if (math.MaxUint64-gas)/params.TxDataZeroGas < z { // @note redundant check
      return 0, ErrGasUintOverflow
    }
    gas += z * params.TxDataZeroGas
  }
  return gas, nil
}
```