



# VeChainThor Galactica Security Assessment

VECHAIN FOUNDATION SAN MARINO SRL  
Version 1.0 – May 6, 2025

©2025 – NCC Group

Prepared by NCC Group Security Services, Inc. for VECHAIN FOUNDATION SAN MARINO SRL. Portions of this document and the templates used in its production are the property of NCC Group and cannot be copied (in full or in part) without NCC Group's permission.

While precautions have been taken in the preparation of this document, NCC Group the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of NCC Group's services does not guarantee the security of a system, or that computer intrusions will not occur.

**Prepared By**  
Kevin Henry  
Parnian Alimi

**Prepared For**  
Neil Brett

# 1 Executive Summary

---

## Synopsis

In April 2025, VeChain engaged NCC Group's Cryptography Services practice to perform an implementation review of several updates culminating in the "Galactica" fork of the VeChain Thor blockchain. VeChainThor is an EVM-compatible layer 1 blockchain which caters to enterprise users and is intended to serve as a foundation for a sustainable and scalable enterprise blockchain ecosystem.

The review was delivered remotely by two consultants in 15 person-days, including retesting. VeChain promptly responded to the findings and comments made in this report, with both findings considered *Fixed* and all non-informational recommendations implemented.

## Scope

The review targeted the [release/galactica](#) branch of the [vechain/thor](#) repository at [commit dda032f](#). In particular, the review was focused on changes from the `master` branch ([commit 855ae30](#) at the time of review), which primarily consisted of the adoption of several [VIPs](#):

- [VIP-242](#): ETH Shanghai Upgrade
- [VIP-251](#): Dynamic Fee Market
- [VIP-250](#): Expose Clause Information
- [VIP-252](#): Typed Transaction Support

In addition to a review of the diff, a targeted review of each VIP (and EIP) added in Galactica was performed to ensure that each VIP was correctly integrated, resulting in several notes and comments as summarized in the [VIP/EIP Review](#) section.

## Limitations

The NCC Group team focused their review on the scope above, paying attention to common programming pitfalls and security issues. Note that the Galactica release adds features to the existing Thor node implementation which was not reviewed in its entirety. The consultants achieved good coverage on the in-scope changes.

## Key Findings

The assessment uncovered 2 findings:

- [Finding "Transactions with the Same Priority Fee Should be Sorted by Age"](#) warns about the possibility of a spamming attack on the transaction pool.
- [Finding "Potential Division-by-Zero May Cause Denial-of-Service"](#) recommends to add comments to warn about the potential division-by-zero panic.

Upon retesting, both findings were *Fixed*, and all non-informational recommendations from the comments were addressed.

## Strategic Recommendations

Overall NCC Group found VeChain's "Galactica" fork of the Thor blockchain to be well-implemented. The code was well-commented, followed preferred usage patterns of underlying libraries, consistently considered unusual error conditions such as integer overflows, and we noted few security concerns overall. Beyond addressing the findings and remarks presented in this review, NCC Group recommends increasing node diversity by incentivizing an independent node implementation. Having multiple implementations strengthens the ecosystem by decreasing the probability of an outage in case a single implementation is vulnerable and is under attack<sup>1</sup>.

---

1. In May of 2023, [Ethereum briefly experienced finality issues](#), due to a bug in the Prysm client (used by 37% of the nodes at the time) which slowed block validation. The remaining nodes in the network continued to produce blocks which ensured liveness.



## 2 Dashboard

### Target Data

Name	VeChainThor
Type	Source Code
Platforms	Golang

### Engagement Data



Type	Implementation Review
Method	Code-Assisted
Dates	2025-04-07 to 2025-04-25
Consultants	2
Level of Effort	15 person-days

### Targets



The Galactica Release Changes

[release/galactica](#) branch at [commit dda032f](#)



### Finding Breakdown






Critical issues	0
High issues	0
Medium issues	0
Low issues	1 
Informational issues	1 
<b>Total issues</b>	<b>2</b>

### Category Breakdown

Denial of Service	1 
Security Improvement Opportunity	1 

### Component Breakdown

General	1 
txpool	1 

 Critical    High    Medium    Low    Informational



### 3 Table of Findings

---

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Transactions with the Same Priority Fee Should be Sorted by Age	Fixed	92C	Low
Potential Division-by-Zero May Cause Denial-of-Service	Fixed	UPN	Info



## 4 Finding Details

Low

# Transactions with the Same Priority Fee Should be Sorted by Age

Overall Risk Low

Impact Low

Exploitability Medium

Finding ID NCC-E023736-92C

Component txpool

Category Security Improvement  
Opportunity

Status Fixed

### Impact

Failure to evict similar transactions based on their age (time passed since they were added to the pool), can be exploited by an attacker to exclude other transactions from the pool.

### Description

The following quote from [EIP-1559](#) (equivalent of [VIP-251](#)) warns about a spamming attack in which an attacker pushes honest transactions out of the pool:

With most people not competing on priority fees and instead using a baseline fee to get included, transaction ordering now depends on individual client internal implementation details such as how they store the transactions in memory. It is recommended that transactions with the same priority fee be sorted by time the transaction was received to protect the network from spamming attacks where the attacker throws a bunch of transactions into the pending pool in order to ensure that at least one lands in a favorable position. Miners should still prefer higher gas premium transactions over those with a lower gas premium, purely from a selfish mining perspective.

The Thor node's `housekeeping()` and in turn `wash()` implementations continuously monitor the transaction pool and prune it if its size exceeds the limit:

```
// wash to evict txs that are over limit, out of lifetime, out of energy, settled, expired or
↳ dep broken.
// this method should only be called in housekeeping go routine
func (p *TxPool) wash(headSummary *chain.BlockSummary) (executables tx.Transactions, removed
↳ int, err error) {
    // ...Initialization...
    // ...Remove out of lifetime, settled, out of energy, or dep broken transactions...

    // sort objs by price from high to low.
    sortTxObjsByOverallGasPriceDesc(executableObjs)

    limit := p.options.Limit

    // remove over limit txs, from non-executables to low priced
    if len(executableObjs) > limit {
        for _, txObj := range nonExecutableObjs {
            toRemove = append(toRemove, txObj)
            logger.Debug("non-executable tx washed out due to pool limit", "id", txObj.ID())
        }
        for _, txObj := range executableObjs[limit:] {
```



```

    toRemove = append(toRemove, txObj)
    logger.Debug("executable tx washed out due to pool limit", "id", txObj.ID())
}
executableObjs = executableObjs[:limit]

```

Figure 1: Excerpt from `wash()` in `txpool/tx_pool.go`

The `sortTxObjsByOverallGasPriceDesc()` function, depicted below, sorts the transaction objects based on their `priorityGasPrice`. Note that Golang's `sort.Slice()` implementation is *not guaranteed to be stable*. Thereby, after calling `sortTxObjsByOverallGasPriceDesc()` the transactions' ordering might change and transactions with smaller `timeAdded` (i.e., older transactions) may not be in the lower indices. As a result the last highlighted line above, might remove older transactions (with the same priority fee) first.

```

124 func sortTxObjsByOverallGasPriceDesc(txObjs []*txObject) {
125     sort.Slice(txObjs, func(i, j int) bool {
126         gp1, gp2 := txObjs[i].priorityGasPrice, txObjs[j].priorityGasPrice
127         return gp1.Cmp(gp2) >= 0
128     })
129 }

```

Figure 2: `txpool/tx_object.go`

## Recommendation

Update the `sortTxObjsByOverallGasPriceDesc()` function to sort the transaction objects based on their `timeAdded` when their `priorityGasPrice` is the same.

## Location

[https://github.com/vechain/thor/blob/dda032fae6bebdb6e8302edfc7d0662f20c468d9/txpool/tx\\_object.go#L124-L129](https://github.com/vechain/thor/blob/dda032fae6bebdb6e8302edfc7d0662f20c468d9/txpool/tx_object.go#L124-L129)

## Retest Results

### 2025-04-30 – Fixed

In [commit 582aca6](#), this finding's recommendation was implemented, and also a unit test was included. This finding was marked *Fixed* as a result.

# Potential Division-by-Zero May Cause Denial-of-Service

Overall Risk Informational

Impact Low

Exploitability Low

Finding ID NCC-E023736-UPN

Category Denial of Service

Status Fixed

## Impact

Failure to prevent division-by-zero causes a panic in the Big Integer library, which in turn crashes the node application.

## Description

The Golang's big integer math library warns that division-by-zero results in a runtime panic<sup>2</sup>. As such it is prudent to ensure that the denominator is non-zero prior to a division operation. Thor's transactions' overall gas price calculation and the consensus module's base fee calculation logics do not explicitly perform this check. As a result, if this corner case is not caught by the upper layer a malicious actor may craft a transaction that will crash the Thor node.

```

491 // OverallGasPrice calculate overall gas price.
492 // overallGasPrice = gasPrice + baseGasPrice * wgas/gas.
493 func (t *Transaction) OverallGasPrice(baseGasPrice *big.Int, provedWork *big.Int) *big.Int
494 ↪ {
495     gasPrice := t.GasPrice(baseGasPrice)
496
497     if provedWork.Sign() == 0 {
498         return gasPrice
499     }
500
501     wgas := workToGas(provedWork, t.BlockRef().Number())
502     if wgas == 0 {
503         return gasPrice
504     }
505     if wgas > t.body.gas() {
506         wgas = t.body.gas()
507     }
508
509     x := new(big.Int).SetUint64(wgas)
510     x.Mul(x, baseGasPrice)
511     x.Div(x, new(big.Int).SetUint64(t.body.gas()))
512     return x.Add(x, gasPrice)
513 }
```

Figure 3: tx/transaction.go

NCC Groups' attempts at triggering this panic from higher in the call stack, e.g., the transaction pool, were unsuccessful. Take the `PrepareTransaction()` to the `GalacticaPriorityPrice()`, to the `OverallGasPrice()` chain call for example. A transaction that is finalized by the `PrepareTransaction()` function, is **resolved first** which ensures that the transaction's gas is not below its intrinsic gas cost. Since the minimum intrinsic gas cost is larger than 0 (currently 21000), a transaction that has its gas set to 0, is rejected with the "intrinsic gas

2. <https://pkg.go.dev/math/big#Int.Div>

---

*exceeds provided gas*” error. Thereby such malformed transactions will not be finalized and the panic will not be triggered in reality. As such the severity of this finding is set to informational, and implementing the recommended measure is considered a security improvement opportunity in the event future development does not consider this corner case.

## Recommendation

Consider adding a check to ensure that the denominator is not 0 prior to a big integer division. Alternatively, document this corner case to avoid unexpected behavior in the future.

## Location

- <https://github.com/vechain/thor/blob/dda032fae6bebdb6e8302edfc7d0662f20c468d9/tx/transaction.go#L510>
- <https://github.com/vechain/thor/blob/dda032fae6bebdb6e8302edfc7d0662f20c468d9/consensus/fork/galactica.go#L76>

## Retest Results

### 2025-04-30 – Fixed

In `commit acec779`, code comments were added to explain that division-by-zeros are not possible due to the fact that the intrinsic gas pre-check prevents them. This finding was marked *Fixed* as a result.





## 5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

### Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

### Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

### Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

### Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.



Rating	Description
<b>Medium</b>	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
<b>Low</b>	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

## Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
<b>Access Controls</b>	Related to authorization of users, and assessment of rights.
<b>Auditing and Logging</b>	Related to auditing of actions, or logging of problems.
<b>Authentication</b>	Related to the identification of users.
<b>Configuration</b>	Related to security configurations of servers, devices, or software.
<b>Cryptography</b>	Related to mathematical protections for data.
<b>Data Exposure</b>	Related to unintended exposure of sensitive information.
<b>Data Validation</b>	Related to improper reliance on the structure or values of data.
<b>Denial of Service</b>	Related to causing system failure.
<b>Error Reporting</b>	Related to the reporting of error conditions in a secure fashion.
<b>Patching</b>	Related to keeping software up to date.
<b>Session Management</b>	Related to the identification of authenticated users.
<b>Timing</b>	Related to race conditions, locking, or order of operations.



## 6 VIP/EIP Review

One of the core areas under review was the implementation and integration of several VIPs. No security findings relating to these VIPs were uncovered. A small number of comments and recommendations are documented here, alongside responses from the VeChain team, as well as positive evidence of correct implementation. In summary:

- A test for maximum stack size (1024) as specified in EIP-3855 is not present, but the correct behavior was manually validated.
  - **Retest Update:** As part of [PR 1058](#) code comments were updated to point to the enforcement of the maximum stack size.
- The `ModExp` gas cost calculation was updated according to EIP-2565. The NCC reviewers note that there is a later proposal to increase the minimum gas cost of this operation from 200 to 500; see [EIP-7883](#).
  - **Retest Update:** The linked EIP is still in draft state and is not yet recommended for Ethereum. As such, it is also not being adopted by VeChain Thor at this time.
- For dynamic fee transactions, there exists a corner case where the gas limit calculation could theoretically overflow, although this does not appear to be a practical concern.
  - **Retest Update:** As no exploitable case was identified, no changes were made.
- For VeChain-specific transaction types, defined in VIP-252, it is recommended to make the inclusion of the transaction type byte in the signing input a “MUST” requirement instead of a “SHOULD” requirement.
  - **Retest Update:** As part of [PR 88](#), the VIP was updated to make this a “MUST” requirement.

Additional details can be found in the relevant subsections.

### VIP-242: ETH Shanghai Upgrade Specification

[VIP-242](#) specifies a list of EIPs added to VeChain:

This VIP specifies the changes required to align the EVM with the Shanghai release of Ethereum.

No issues were found with the various integrations.

### EIP-3198: BASEFEE opcode

This EIP adds a new opcode to return the base fee of the contract:

```
799 // opBaseFee implements BASEFEE opcode
800 func opBaseFee(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
    ↳ ([]byte, error) {
801     baseFee, _ := uint256.FromBig(evm.Context.BaseFee)
802     stack.push(baseFee)
803     return nil, nil
804 }
```

Figure 4: Excerpt from [vm/instructions.go](#)

This opcode is added to the jump table and is tested alongside similar functions as expected.

[EIP-3198](#) notes the following:

The value of the base fee is not sensitive and is publicly accessible in the block header. There are no known security implications with this opcode.

In line with this comment, no security concerns were found.

### EIP-3855: PUSH0 instruction

This EIP adds a new opcode that pushes 0 to the stack:

```
806 // opPush0 implements the PUSH0 opcode
807 func opPush0(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
    ↳ ([]byte, error) {
808     stack.push(new(uint256.Int))
809     return nil, nil
810 }
```

Figure 5: Excerpt from [vm/instructions.go](#)

EIP-3855 notes the following:

The authors are not aware of any impact on security. Note that jumpdest-analysis is unaffected, as PUSH0 has no immediate data bytes.

In line with this comment, no security concerns were found.

The EIP does specify some test cases, including calling the opcode 1025 times and ensuring a stack overflow error. However, the mock stack implementation does not enforce this limit, and such a test case would pass in the reviewed test suites. While reviewing this behavior, a stale comment was observed:

```
53 func (st *Stack) push(d *uint256.Int) {
54     // NOTE push limit (1024) is checked in baseCheck
55     st.data = append(st.data, *d)
56 }
```

Figure 6: Excerpt from [vm/stack.go](#)

The actual enforcement of the maximum stack size occurs as part of `call()` in [vm/evm.go](#) starting on line 193.

### Retest Update

As part of [PR 1058](#) code comments were updated to point to the enforcement of the maximum stack size.

### EIP-3541: Reject new contracts starting with the 0xEF byte

As title of this [EIP-2541](#) implies, contracts starting with 0xEF should be rejected:

```
494 // Reject code starting with 0xEF if EIP-3541 is enabled.
495 if err == nil && len(ret) >= 1 && ret[0] == 0xEF && evm.chainRules.IsShanghai {
496     err = ErrInvalidCode
497 }
```

Figure 7: Excerpt from `create()` in [vm/evm.go](#)

This behavior is also tested in the runtime tests, with the pre-update behavior validated, and 5 test cases from the EIP correctly implemented.

The EIP also notes:

The authors are not aware of any security or DoS risks posed by this change.

In line with this comment, no security concerns were found.

## EIP-1108: Reduce alt\_bn128 precompile gas costs

EIP-1108 defines updated gas costs for certain elliptic curve operations:

```
36 Bn256AddGasEIP1108          uint64 = 150 // Gas needed for an elliptic curve addition
37 Bn256ScalarMulGasEIP1108    uint64 = 6000 // Gas needed for an elliptic curve scalar
    ↳ multiplication
38 Bn256PairingBaseGasEIP1108  uint64 = 45000 // Base price for an elliptic curve pairing
    ↳ check
39 Bn256PairingPerPointGasEIP1108 uint64 = 34000 // Per-point price for an elliptic curve
    ↳ pairing check
```

Figure 8: [vm/gas.go](#)

These values will be returned for all contracts using the Shanghai instruction set, as implemented in [vm/contracts.go](#).

## EIP-2565: ModExp Gas Cost

EIP-2565 provides a more accurate mechanism for estimating the gas cost of the `ModExp` operation, while instituting a minimum cost of 200.

The corresponding `RequiredGas()` gas function has been updated to reflect this behavior, which matches the implementation in `go-ethereum`; see [PR 21607](#). Notably, as discussed in the linked PR, the implementation is structured differently than the presentation in EIP-2565, which makes aligning their behavior unnecessarily difficult.

The `eip2565` flag is set to true in the Shanghai instruction set, ensuring the updated cost is used.

The EIP notes:

The biggest security consideration for this EIP is creating a potential DoS vector by making `ModExp` operations too inexpensive relative to their computation time.

While the original EIP includes a data plot demonstrating the accuracy of the updated approach, this concern appears to be valid, and there exists a proposal in [EIP-7883](#) to increase the minimum gas cost from 200 to 500.

## Retest Update

The linked EIP is still in draft state and is not yet recommended for Ethereum. As such, it is also not being adopted by VeChain Thor at this time.

## VIP-250: Extend EVM's context to expose clause information

VIP-250 defines the straightforward addition to expose additional context information:

outlines an upgrade to the EVM, which will expose clause information to the smart contract, via the built-in extension contract.

The updates to the extension contract in the VIP are included in [builtin/gen/extension-v3.sol](#) verbatim, with tests included in `TestExtensionV3()` in [builtin/native\\_calls\\_test.go](#).

## VIP-251: Dynamic Fee Market

VIP-251 introduces the following:

proposes implementing a dynamic fee mechanism for VeChainThor transactions, inspired by Ethereum's EIP-1559. It introduces a fluctuating base fee that is burned and a user-set priority fee that is paid directly to the validator proposing the block.

The implemented logic is relatively straightforward, with each block having a target gas limit based on its parent. Blocks exceeding their gas target will increase fees, and blocks that do not reach their target will reduce fees, with all changes based on an elasticity modifier to ensure changes are gradual.

These changes are primarily implemented as `CalcBaseFee()` in [consensus/fork/galactica.go](#) with helper functions in [block/gas\\_limit.go](#).

The implemented behavior appears to match the VIP, however the following was noted:

```
59  var (  
60      parentGasTarget          = parent.GasLimit() * thor.ElasticityMultiplierNum / thor.Elast  
        ↳ icityMultiplierDen  
61      parentGasTargetBig       = new(big.Int).SetUint64(parentGasTarget)  
62      baseFeeChangeDenominator = new(big.Int).SetUint64(thor.BaseFeeChangeDenominator)  
63  )
```

Figure 9: Excerpt from `CalcBaseFee()` in [consensus/fork/galactica.go](#)

Here, the value `parentGasTarget` is computed as a `uint64`, which involves the product `parent.GasLimit() * thor.ElasticityMultiplierNum`, currently `parent.GasLimit() * 3`. Should the parent gas limit grow large enough, this value could potentially overflow. Such an occurrence is extremely unlikely given the initial state and expected behavior:

```
29  InitialGasLimit    uint64 = 10 * 1000 * 1000 // InitialGasLimit gas limit value int  
        ↳ genesis block.  
30  GasLimitBoundDivisor uint64 = 1024           // from ethereum
```

Figure 10: Excerpt from [thor/params.go](#)

The dynamic fee behavior is extensively tested in [consensus/fork/galactica\\_test.go](#).

## Retest Update

As no exploitable case for integer overflow was identified, no changes were made.

## VIP-252: Typed Transaction Support

VIP-252 mirrors EIP-2718:

This VIP proposes a new transaction envelope format that increases flexibility and forward compatibility for the VeChainThor blockchain.

This enables the adoption of a dynamic fee transaction while retaining backwards compatibility with legacy transactions. At the time of review, this is the only additional transaction type implemented, with an internal type identifier of `0x51`.

Both VIP-252 and EIP-2718 specify a requirement that the type identifier SHOULD be included in any signatures but does not make this a MUST requirement. The implemented dynamic fee transaction does include this value, as recommended:

```
284 // SigningHash returns hash of tx excludes signature.  
285 func (t *Transaction) SigningHash() (hash thor.Bytes32) {  
286     if cached := t.cache.signingHash.Load(); cached != nil {  
287         return cached.(thor.Bytes32)  
288     }  
289     defer func() { t.cache.signingHash.Store(hash) }()  
290  
291     if t.Type() == TypeLegacy {  
292         return rlpHash(t.body.signingFields())
```



```
293 }  
294 // Include type prefix for typed tx.  
295 return prefixedRlpHash(t.Type(), t.body.signingFields())  
296 }
```

Figure 11: Excerpt from [tx/transaction.go](#)

This ensures that a signature from another transaction type cannot pass as a dynamic fee transaction signature. Similar logic is in place to ensure the type is included for `EncodeRLP()`, used when creating the `ReceiptsHash`. Note that this later use case is a MUST requirement in VIP-252.

Consider the requirement in VIP-252:

As of `FORK_BLOCK_NUMBER`, the receipt root in the block header MUST be the root hash of `patriciaTrie(rlp(Index) => Receipt)`

This suggests that for VeChain-specific transaction types (e.g., `0x00 .. 0x7F`), it would be advisable to make the corresponding requirement to sign the transaction type a “MUST” requirement as well. Such a change is beneficial as a defense in depth measure, as it would ensure that a signed dynamic fee transaction cannot potentially be used for any future transaction type that might omit the type identifier in its hash.

#### Retest Update

As part of [PR 88](#), the VIP was updated to make this a “MUST” requirement.