SOFTWARE AUDIT REPORT

for

AVA LABS

Prepared By: Shuxiao Wang

Hangzhou, China

Jun. 10, 2020

Document Properties

Client	AVA Labs
Title	Software Audit Report
Target	Avalanche Blockchain
Version	0.1
Author	Jeff Liu
Auditors	Edward Lo, Ruiyi Zhang, Xudong Shao
Reviewed by	Chiachih Wu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author	Description
0.1	Jun. 10, 2020	Jeff Liu	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Avalanche Blockchain** design document and related source code, we in this report outline our systematic method to evaluate potential security issues in the Avalanche Blockchain implementation, expose possible semantic inconsistencies between the source code and the design specification, and provide additional suggestions and recommendations for improvement. Our results show that the given branch of Avalanche Blockchain can be further improved due to the presence of several issues related to either security or performance. This document describes our audit results in detail.

1.1 About Avalanche Blockchain

Avalanche Blockchain (AVA) [1] is a public blockchain system designed by AVA Labs [2]. AVA is a high-throughput, finance-focused, open-source blockchain system supporting highly decentralized applications and financial primitives, also interoperable blockchains. It can achieve 4000+ TPS with less than 3 seconds transaction finality time, mainly becuase of its DAG-optimized consensus protocol, and utilizing interoperable subnets to facilitate high scalability.

The basic information of Avalanche Blockchain is shown in Table 1.1, and its Git repository and the commit hash value (of the audited branch) are in Table 1.2.

Table 1.1: Basic Information of Avalanche Blockchain

Item	Description
Issuer	AVA Labs
Website	https://avalabs.org
Type	AVA Blockchain
Platform	Go
Audit Method	White-box
Latest Audit Report	Jun. 10, 2020

Table 1.2: The Commit Hash List Of Audited Branches

Git Repository	Commit Hash Of Audited Branch
github.com/ava-labs/gecko-security-analysis	f66fd6de584e3e0347d39a62d427c440936ab745

1.2 About PeckShield

PeckShield Inc. [3] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products including security audits. We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

In the first phase of auditing Avalanche Blockchain, we use fuzzing to find out the corner cases that may not be covered by in-house testing. Next we do white-box auditing, in which PeckShield security auditors manually review Avalanche Blockchain design and source code, analyze them for any potential issues, and follow up with issues found in the fuzzing phase. If necessary, we design and implement individual test cases to further reproduce and verify the issues. In the following subsections, we will introduce the risk model as well as the audit procedure adopted in this report.

Table 1.3: Vulnerability Severity Classification

Impact \ **Likelihood**	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3.1 Risk Model

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [4]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, and *Low* shown in Table 1.3.

1.3.2 Fuzzing

Fuzzing or fuzz testing is an automated software testing technique of discovering software vulnerabilities by systematically finding and providing possible inputs to the target program, and then monitoring the program execution for crashes (or any unexpected results). In the first phase of our audit, we use fuzzing to find out possible corner cases or unusual inter-module interactions that may not be covered by in-house testing. As one of the most effective methods for exposing the presence of possible vulnerabilities, fuzzing technology has been the first choice for many security researchers in recent years. At present, there are many fuzzy testing tools and supporting software, which can help security personnels to conduct fuzzing and find vulnerabilities more efficiently. Based on the characteristics of the Avalanche Blockchain, we use AFL [5] as the primary tool for fuzz testing.

AFL (American Fuzzy Lop) is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. Since its inception, AFL has gained growing popularity in the industry and has proved its effectiveness in discovering quite a few significant software bugs in a wide range of major software projects. The basic process of AFL fuzzing is as follows:

- Generate compile-time instrumentation to record information such as code execution path;
- Construct some input files to join the input queue, and change input files according to different strategies;
- Files that trigger a crash or timeout when executing an input file are logged for subsequent analysis;
- Loop through the above process.

Throughout the AFL testing, we will reproduce each crash based on the crash file generated by AFL. For each reported crash case, we will further analyze the root cause and check whether it is indeed a vulnerability. Once a crash case is confirmed as a vulnerability of the Avalanche Blockchain, we will further analyze it as part of the white-box audit.

1.3.3 White-box Audit

After fuzzing, we continue the white-box audit by manually analyzing source code. Here we test target software's internal structure, design, coding, and we focus on verifying the flow of input and output through the application as well as examining possible design and implementation trade-offs for strengthened security. PeckShield auditors first fully review and understand the source code, then create specific test cases, execute them and analyze the results. Issues such as internal security loopholes, unexpected output, broken or poorly structured paths, etc., will be inspected under close scrutiny.

Blockchain is a secure method of creating a distributed database of transactions, and three major technologies of blockchain are cryptography, decentralization, and consensus model. Blockchain does come with unique security challenges, and based on our understanding of blockchain general design, we in this audit divide the blockchain software into the following major areas and inspect each area accordingly:

- Data and state storage, which is related to the database and files where blockchain data are saved.
- P2P networking, consensus, and transaction model in the networking layer. Note that the consensus and transaction logic is tightly coupled with networking.
- VM, account model, and incentive model. This is essentially the execution and business layer of the blockchain, and many blockchain business specific logics are implemented here.
- System contracts and services. These are system-level, blockchain-wide operation management contracts and services.
- Others. This includes any software modules that do not belong to above-mentioned areas, such as common crypto or other 3rd-party libraries, best practice or optimization used in other software projects, design and coding consistency, etc.

Based on the above classification, we show in Table 1.4 and Table 1.5 the detailed list of the audited items in this report.

To better describe each issue we identified, we also categorize the findings based on Common Weakness Enumeration (CWE-699) [6], which is a community-developed list of software weakness types to better classify and organize weaknesses around concepts frequently encountered in software development. We use the CWE categories in Table 1.6 to classify our findings.

Table 1.4: The Full List of Audited Items (Part I)

Category	Check Item
Data and State Storage	Blockchain Database Security
	Database State Integrity Check
Node Operation	Default Configuration Security
	Default Configuration Optimization
	Node Upgrade And Rollback Mechanism
Node Communication	External RPC Implementation Logic
	External RPC Function Security
	Node P2P Protocol Implementation Logic
	Node P2P Protocol Security
	Serialization/Deserialization
	Invalid/Malicious Node Management Mechanism
	Communication Encryption/Decryption
	Eclipse Attack Protection
	Fingerprint Attack Protection
Consensus	Consensus Algorithm Scalability
	Consensus Algorithm Implementation Logic
	Consensus Algorithm Security
Transaction Model	Transaction Privacy Security
	Transaction Fee Mechanism Security
	Transaction Congestion Attack Protection
VM	VM Implementation Logic
	VM Implementation Security
	VM Sandbox Escape
	VM Stack/Heap Overflow
	Contract Privilege Control
	Predefined Function Security
Account Model	Status Storage Algorithm Adjustability
	Status Storage Algorithm Security
	Double Spending Protection
Incentive Model	Mining Algorithm Security
	Mining Algorithm ASIC Resistance
	Tokenization Reward Mechanism

Table 1.5: The Full List of Audited Items (Part II)

Category	Check Item
System Contracts And Services	Memory Leak Detection
	Use-After-Free
	Null Pointer Dereference
	Undefined Behaviors
	Deprecated API Usage
	Signature Algorithm Security
	Multisignature Algorithm Security
SDK Security	Using RPC Functions Security
	Privatekey Algorithm Security
	Communication Security
	Function integrity checking code
Others	Third Party Library Security
	Memory Leak Detection
	Exception Handling
	Log Security
	Coding Suggestion And Optimization
	White Paper And Code Implementation Uniformity

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of blockchain software. Last but not least, this security audit should not be used as an investment advice.

Table 1.6: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Input Validation Issues	Weaknesses in this category are related to a software system's input validation components. Frequently these deal with sanitizing, neutralizing and validating any externally provided inputs to minimize malformed data from entering the system and preventing code injection in the input data.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Finding Summary

Here is a summary of our findings after analyzing Avalanche Blockchain. As mentioned earlier, we in the first phase of our audit studied AVA source code and ran our in-house static code analyzer through the codebase, and the purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tools. After that, we manually review business logics, examine system operations, and place operation-specific aspects under scrutiny to uncover possible pitfalls and/or bugs.

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple modules. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined more than a dozen of issues of varying severities that need to be brought up, which are categorized in Table 2.1. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

Table 2.1: The Severity of Our Findings

Severity	# of Findings	
Critical	14	■■■■■■■■■■■■■■
High	2	■■
Medium	2	■■
Low	0	
Informational	0	
Total	18	

2.2 Key Findings

After analyzing all of the potential issues found during the audit, we determined that a number of them need to be brought up and paid more attention to, as shown in Table 2.2. Please refer to Section 3 for detailed discussion of each issue.

Table 2.2: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	DoS Vulnerability in the VM Module - #1	Coding Practices	Fixed
PVE-002	Critical	DoS Vulnerability in the VM Module - #2	Business Logic Errors	Confirmed
PVE-003	Critical	DoS Vulnerability in the VM Module - #3	Business Logic Errors	Confirmed
PVE-004	Critical	DoS Vulnerability in the VM Module - #4	Business Logic Errors	Confirmed
PVE-005	High	Out-of-Memory Issue in the VM Module	Coding Practices	Fixed
PVE-006	Critical	Missing Sanity Check in the Avalanche Consensus Process - #1	Coding Practices	Confirmed
PVE-007	Critical	Missing Sanity Check in the Avalanche Consensus Process - #2	Coding Practices	Confirmed
PVE-008	Critical	DoS Vulnerability in the Avalanche Consensus Module - #1	Coding Practices	Confirmed
PVE-009	Critical	DoS Vulnerability in the Avalanche Consensus Module - #2	Coding Practices	Confirmed
PVE-010	Critical	DoS Vulnerability in the Avalanche Consensus Module - #3	Behavioral Problems	Confirmed
PVE-011	Critical	DoS Vulnerability in the Avalanche Consensus Module - #4	Coding Practices	Confirmed
PVE-012	Critical	Missing Sanity Check in the Bootstrapping Process	Input Validation Issues	Fixed
PVE-013	Critical	Missing Sanity Check in the Snowman Consensus Process - #1	Coding Practices	Confirmed
PVE-014	Critical	Missing Sanity Check in the Snowman Consensus Process - #2	Input Validation Issues	Fixed
PVE-015	High	Missing Sanity Check in the RPC Module	Input Validation Issues	Confirmed
PVE-016	Medium	DoS Vulnerability in the RPC Module	Coding Practices	Confirmed
PVE-017	Critical	DoS Vulnerability in the P2P Module	Coding Practices	Confirmed
PVE-018	Medium	DoS Vulnerability in the VM Module - #5	Coding Practices	Confirmed

3 Detailed Results

3.1 DoS Vulnerability in the VM Module - #1

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `snow/validators/sampler.go`
- Category: Coding Practices [7]
- CWE subcategory: CWE-20 [8]

Description

There is a vulnerability in the platformvm module of AVA, which could be exploited by attackers to perform DoS attack.

```go
// SampleValidatorsArgs are the arguments for calling SampleValidators
type SampleValidatorsArgs struct {
    Size int `json:"size"`
}

// SampleValidatorsReply are the results from calling Sample
type SampleValidatorsReply struct {
    Validators []string `json:"validators"`
}

// SampleValidators returns a sampling of the list of current validators
func (service *Service) SampleValidators(_ *http.Request, args *SampleValidatorsArgs, reply *SampleValidatorsReply) error {
    service.vm.ctx.Log.Debug("Sample called with {Size = %d}", args.Size)

    validatorSet := service.vm.Validators.Sample(args.Size)
```

Listing 3.1: vms/platformvm/service.go

This API allows clients to interact with the P-Chain (Platform Chain), which maintains AVA's validator set and handles blockchain creation.

`Platform.SampleValidators` responds with sample `size` validators from the validator set of the default subnet.

```go
139 // Sample implements the Group interface.
140 func (s *Sampler) Sample(size int) []Validator {
141     s.lock.Lock()
142     defer s.lock.Unlock()
143
144     return s.sample(size)
145 }
146
147 func (s *Sampler) sample(size int) []Validator {
148     list := make([]Validator, size)[:0]
```

Listing 3.2: snow/validators/sampler.go

`sample` creates a slice for SampleValidators(line 148). However, it would lead to a panic (runtime error: makeslice: len out of range) when parameter `size` is negative.

Recommendation Add a sanity check for the parameter `size`.

3.2 DoS Vulnerability in the VM Module - #2

- ID: PVE-002
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `vms/spchainvm/block.go`
- Category: Business Logic Errors [9]
- CWE subcategory: CWE-770 [10]

Description

There is a vulnerability in gecko's spchainvm module, which could be exploited by attackers to perform DoS attack against the AVA network.

```go
// ParseBlock implements the snowman.ChainVM interface
func (vm *VM) ParseBlock(b []byte) (snowman.Block, error) {
    c := Codec{}
    rawBlock, err := c.UnmarshalBlock(b)
    if err != nil {
        return nil, err
    }
    block := &LiveBlock{
        vm:    vm,
        block: rawBlock,
    }
    if err := block.VerifyBlock(); err != nil {
        return nil, err
    }
    return block, vm.state.SetBlock(vm.baseDB, rawBlock.ID(), rawBlock)
}
```

Listing 3.3: snow/engine/snowman/transitive.go

`ParseBlock` calls `unmarshalBlock` to deserialize the block, then check the validity of it by `block.VerifyBlock()`.

```
// VerifyBlock the validity of this block
func (lb *LiveBlock) VerifyBlock() error {
    if lb.verifiedBlock {
        return lb.validity
    }

    lb.verifiedBlock = true
    lb.validity = lb.block.verify(lb.vm.ctx, &lb.vm.factory)
    return lb.validity
}
```

Listing 3.4: vms/spchainvm/live_block.go

```
func (b *Block) verify(ctx *snow.Context, factory *crypto.FactorySECP256K1R) error {
    switch {
    case b == nil:
        return errInvalidNil
    case b.id.IsZero():
        return errInvalidID
    case b.parentID.IsZero():
        return errInvalidID
    }

    for _, tx := range b.txs {
        if err := tx.verify(ctx, factory); err != nil {
            return err
        }
    }
    return nil
}
```

Listing 3.5: vms/spchainvm/block.go

However, there is no check on sequence of transactions in blocks. An attacker could craft lots of blocks with valid contents(same transactions but in different order). `UnmarshalBlock` would successfully deserialize the blocks and calculate a different id. In the end, `SetBlock` would store them into database, which could cause the victim nodes Out-of-Storage.

Furthermore, the victims would pass the blocks to other sampled validators by calling `PushQuery`, and the new victims would repeat this process, resulting in a DDoS attack eventually.

Recommendation Preventing nodes from flooding the network with blocks/vertices.

3.3 DoS Vulnerability in the VM Module - #3

- ID: PVE-003
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `vms/platformvm/prefixed_state.go`
- Category: Business Logic Errors [9]
- CWE subcategory: CWE-770 [10]

Description

There is a vulnerability in gecko's platformvm module, which could be exploited by attackers to perform DoS attack against the AVA network.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

A node will send a Put message in response to receiving a Get message for a container the node has access to.

```go
    func (t *Transitive) Put(vdr ids.ShortID, requestID uint32, blkID ids.ID, blkBytes
        []byte) {
        blk, err := t.Config.VM.ParseBlock(blkBytes)
        if err != nil {
            t.Config.Context.Log.Warn("ParseBlock failed due to %s for block:\n%s",
                err,
                formatting.DumpBytes{Bytes: blkBytes})
            t.GetFailed(vdr, requestID, blkID)
            return
        }

        t.insertFrom(vdr, blk)
    }
```

Listing 3.6: snow/engine/snowman/transitive.go

`Put` handles an incoming Put request from other validators, it will call `VM.ParseBlock` to decode the `Container`.

```go
func (vm *VM) ParseBlock(b []byte) (snowman.Block, error) {
  blk, err := vm.codec.UnmarshalBlock(b)
  if err != nil {
    return nil, err
  }
  if blk, err := vm.getBlock(blk.ID()); err == nil {
    return blk, nil
  }
  vm.state.SetBlock(vm.baseDB, blk)
  return blk, nil
}

```

```
287 // GetBlock implements the snowman.ChainVM interface
288 func (vm *VM) GetBlock(blkID ids.ID) (snowman.Block, error) { return vm.getBlock(blkID) }
289
290 func (vm *VM) getBlock(blkID ids.ID) (Block, error) {
291   if blk, exists := vm.currentBlocks[blkID.Key()]; exists {
292     return blk, nil
293   }
294   if blk := vm.state.Block(vm.baseDB, blkID); blk != nil {
295     return blk, nil
296   }
297   return nil, errMissingBlock
298 }
```

Listing 3.7: vms/platformvm/vm.go

```
    func (s *prefixedState) SetBlock(db database.Database, block Block) {
        s.state.SetBlock(db, s.uniqueID(block.ID(), blockID, s.block), block)
    }
```

Listing 3.8: vms/platformvm/prefixed_state.go

`ParseBlock` calls `getBlock` to look up stored blocks by `blkID` (line 291, 294), then save the returned block into database (line 283)

However, there is not enough check on incoming blocks. An attacker could craft lots of blocks with valid contents, `UnmarshalBlock` would successfully deserialize the blocks and `SetBlock` would store them into database, which could cause the victim nodes Out-of-Storage.

Furthermore, the victims would pass the blocks to other sampled validators by calling `PushQuery`, and the new victims would repeat this process, resulting in a DDoS attack eventually.

Recommendation Preventing nodes from flooding the network with blocks/vertices.

3.4 DoS Vulnerability in the VM Module - #4

- ID: PVE-004
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `vms/spchainvm/block.go`
- Category: Business Logic Errors [9]
- CWE subcategory: CWE-770 [10]

Description

There is a vulnerability in gecko's spchainvm module, which could be exploited by attackers to perform DoS attack against the AVA network.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

A node will send a Put message in response to receiving a Get message for a container the node has access to.

```
func (t *Transitive) Put(vdr ids.ShortID, requestID uint32, blkID ids.ID, blkBytes
    []byte) {
    blk, err := t.Config.VM.ParseBlock(blkBytes)
    if err != nil {
        t.Config.Context.Log.Warn("ParseBlock failed due to %s for block:\n%s",
            err,
            formatting.DumpBytes{Bytes: blkBytes})
        t.GetFailed(vdr, requestID, blkID)
        return
    }

    t.insertFrom(vdr, blk)
}
```

Listing 3.9: snow/engine/snowman/transitive.go

`Put` handles an incoming Put request from other validators, it would call `VM.ParseBlock` to decode the `Container`.

```
// ParseBlock implements the snowman.ChainVM interface
func (vm *VM) ParseBlock(b []byte) (snowman.Block, error) {
    c := Codec{}
    rawBlock, err := c.UnmarshalBlock(b)
    if err != nil {
        return nil, err
    }
    block := &LiveBlock{
        vm:    vm,
        block: rawBlock,
    }
    if err := block.VerifyBlock(); err != nil {
        return nil, err
    }
    return block, vm.state.SetBlock(vm.baseDB, rawBlock.ID(), rawBlock)
}
```

Listing 3.10: snow/engine/snowman/transitive.go

```
func (b *Block) verify(ctx *snow.Context, factory *crypto.FactorySECP256K1R) error {
    switch {
    case b == nil:
        return errInvalidNil
    case b.id.IsZero():
        return errInvalidID
    case b.parentID.IsZero():
        return errInvalidID
    }
```

```go

        for _, tx := range b.txs {
            if err := tx.verify(ctx, factory); err != nil {
                return err
            }
        }
        return nil
    }
```

Listing 3.11: vms/spchainvm/block.go

`ParseBlock` calls `UnmarshalBlock` to deserialize the block, then check the validity of it by `VerifyBlock` (line 166), and finally calls `SetBlock` to store it into database.

The problem is that there are not enough sanity checks on the incoming blocks. An attacker could craft lots of blocks with valid contents, `UnmarshalBlock` would successfully deserialize the blocks and `VerifyBlock` could be easily bypassed if a crafted block has a non-zero parentID and zero txs. In the end, `SetBlock` would store them into database, which could cause the victim nodes Out-of-Storage.

Furthermore, the victims would pass the blocks to other sampled validators by calling `PushQuery` , and the new victims would repeat this process, resulting in a DDoS attack eventually.

Recommendation Preventing nodes from flooding the network with blocks/vertices.

3.5 Out-of-Memory Issue in the VM Module

- ID: PVE-005
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `vms/platformvm/staker.go`
- Category: Coding Practices [7]
- CWE subcategory: CWE-20 [8]

Description

This is a vulnerability in gecko's platformvm module.

```go
    // Verify that this transaction is well formed
    func (tx *AddStakerTx) Verify(networkID uint32) error {
        switch {
        case tx == nil:
            return errNilTx
        case tx.key != nil:
            return nil // Only verify the transaction once
        case tx.id.IsZero():
            return errInvalidID
        case tx.networkID != networkID:
            return errWrongNetworkID
        }
```

```go

        if err := tx.staker.Verify(); err != nil {
            return err
        }
```

Listing 3.12: vms/platformvm/add_staker_tx.go

AddStakerTx is a proposal to add staker to the staking.

```go
60    // Verify that this staker is well formed
61    func (staker *Staker) Verify() error {
62        switch {
63        case staker == nil:
64            return errNilInvalid
65        case staker.id.IsZero():
66            return errInvalidID
67        case staker.destination.IsZero():
68            return errInvalidDestination
69        case staker.amount == 0:
70            return errStakeAmount
71        case staker.endTime <= staker.startTime:
72            return errStakeDuration
73        default:
74            return nil
75        }
76    }
```

Listing 3.13: vms/platformvm/staker.go

`staker.Verify()` checks whether `staker.endTime` is greater than `staker.startTime` (line 71) but ignores the restrictions on duration. Also, we should ensure the proposed staker amount is greater than a Minimum value.

Specifically, a malicious attacker could flood these validators by intentionally proposing AddStakerTx with a large `staker.endTime` and 1 `amount`. After such a transaction is accepted, the new staker would not be removed in the future.

Also, a malicious attacker could flood these validators by intentionally proposing AddStakerTx with a large `staker.endTime` and 1 `amount`. The validators would keep storing these txs and eventually cause them Out-of-Storage.

Recommendation Add sanity checks for these parameters.

3.6 Missing Sanity Check in the Avalanche Consensus Process - #1

- ID: PVE-006
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `snow/engine/avalanche/polls.go`
- Category: Coding Practices [7]
- CWE subcategory: CWE-20 [8]

Description

There is a vulnerability in gecko's Avalanche consensus module, which could be exploited by attackers to compromise the AVA network consensus.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

A node would send a Chits message in response to receiving a PullQuery or PushQuery message for a container the node has added to consensus.

```go
// Chits implements the Engine interface
func (t *Transitive) Chits(vdr ids.ShortID, requestID uint32, votes ids.Set) {
	if !t.bootstrapped {
		t.Config.Context.Log.Warn("Dropping Chits due to bootstrapping")
		return
	}

	v := &voter{
		t:         t,
		vdr:       vdr,
		requestID: requestID,
		response:  votes,
	}
	voteList := votes.List()
	for _, vote := range voteList {
		if !t.reinsertFrom(vdr, vote) {
			v.deps.Add(vote)
		}
	}

	t.vtxBlocked.Register(v)
}
```

Listing 3.14: snow/engine/avalanche/transitive.go

When the vote is registered, `Update` would be called and executes `polls.Vote` to get the final voting results.

```go
func (p *polls) Vote(requestID uint32, vdr ids.ShortID, votes []ids.ID) (ids.UniqueBag, bool) {
```

```
            p.log.All("Vote. requestID: %d. validatorID: %s.", requestID, vdr)
            poll, exists := p.m[requestID]
            p.log.All("Poll: %+v", poll)
            if !exists {
                    return nil, false
            }

            poll.Vote(votes)
            if poll.Finished() {
                    p.log.All("Poll is finished")
                    delete(p.m, requestID)
                    p.numPolls.Set(float64(len(p.m))) // Tracks performance statistics
                    return poll.votes, true
            }
            p.m[requestID] = poll
            return nil, false
    }
```

Listing 3.15: snow/engine/avalanche/polls.go

However, there is no sanity check on whether the validator has already voted. The `vdr` argument which is supposed to be used for checking the duplicate votes is never used here.

And every time `poll.Vote(votes)` gets called, `p.numPending` would decrease by one. The poll would be finished if `p.numPending` reaches 0.

Specifically, an attacker could vote many times to finish the poll before the victim gets votes from other validators. Since `requestId` is decoded from the message directly, attackers could target any `requestId` it wants to interrupt.

Recommendation Add a sanity check to forbid duplicate votes.

3.7 Missing Sanity Check in the Avalanche Consensus Process - #2

- ID: PVE-007
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `snow/consensus/avalanche/topological.go`
- Category: Coding Practices [7]
- CWE subcategory: CWE-20 [8]

Description

This is a vulnerability in gecko's Avalanche consensus module of gecko, which could be exploited by attackers to compromise the AVA network consensus.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

A node would send a Put message in response to receiving a Get message for a container the node has access to.

```
func (t *Transitive) Put(vdr ids.ShortID, requestID uint32, vtxID ids.ID, vtxBytes
    []byte) {
  t.Config.Context.Log.All("Put called for vertexID %s", vtxID)

  if !t.bootstrapped {
      t.bootstrapper.Put(vdr, requestID, vtxID, vtxBytes)
      return
  }

  vtx, err := t.Config.State.ParseVertex(vtxBytes)
  if err != nil {
      t.Config.Context.Log.Warn("ParseVertex failed due to %s for block:\n%s",
      err,
      formatting.DumpBytes{Bytes: vtxBytes})
        t.GetFailed(vdr, requestID, vtxID)
      return
  }
  t.insertFrom(vdr, vtx)
}
```

Listing 3.16: snow/engine/avalanche/transitive.go

`Put` handles an incoming Put request from other validators, it would call `State.ParseVertex` to decode the `Container`

The `ParseVertex` function here simply checks whether the vertex parentIDs and transactions are sorted and unique. If so, the function insertFrom and insert would be called to pass the vertex to consensus engine.

`ParseVertex` simply checks whether the vertex parentIDs and transactions are sorted and unique. If so, `insertFrom` and `insert` would be called to pass the vertex to consensus engine. When the vertex has Unknown parents, node would try `sendRequest(vdr, parent.ID())` to get it back.

```
func (t *Transitive) insertFrom(vdr ids.ShortID, vtx avalanche.Vertex) bool {
    issued := true
    vts := []avalanche.Vertex{vtx}
    for len(vts) > 0 {
        vtx := vts[0]
        vts = vts[1:]

        if t.Consensus.VertexIssued(vtx) {
            continue
        }
        if t.pending.Contains(vtx.ID()) {
            issued = false
            continue
```

```go
			}

			for _, parent := range vtx.Parents() {
				if !parent.Status().Fetched() {
					t.sendRequest(vdr, parent.ID())
					issued = false
				} else {
					vts = append(vts, parent)
				}
			}

			t.insert(vtx)
		}
		return issued
	}
```

Listing 3.17: snow/engine/avalanche/transitive.go

```go
	for _, tx := range txs {
		for _, dep := range tx.Dependencies() {
			depID := dep.ID()
			if !txIDs.Contains(depID) && !t.Consensus.TxIssued(dep) {
				t.missingTxs.Add(depID)
				i.txDeps.Add(depID)
			}
		}
	}

	t.Config.Context.Log.All("Vertex: %s is blocking on %d vertices and %d transactions"
		, vtxID, i.vtxDeps.Len(), i.txDeps.Len())

	t.vtxBlocked.Register(&vtxIssuer{i: i})
	t.txBlocked.Register(&txIssuer{i: i})
```

Listing 3.18: snow/engine/avalanche/transitive.go

As we can see here, two new issuers would be registered in `vtxBlocked` and `txBlocked`. If they have dependencies which haven't been issued, `t.vtxBlocked` would record that issuer to its corresponding depID key in the `map`. Finally, `Update` would be called.

```go
	func (b *Blocker) Register(pending Blockable) {
		b.init()

		for _, pendingID := range pending.Dependencies().List() {
			key := pendingID.Key()
			(*b)[key] = append((*b)[key], pending)
		}

		pending.Update()
	}
```

Listing 3.19: snow/events/blocker.go

```
35  func (i *issuer) Update() {
36      if i.abandoned || i.issued || i.vtxDeps.Len() != 0 || i.txDeps.Len() != 0 || i.t
            .Consensus.VertexIssued(i.vtx) {
37          return
38      }
39      i.issued = true
40
41      vtxID := i.vtx.ID()
42      i.t.pending.Remove(vtxID)
43
44      for _, tx := range i.vtx.Txs() {
45          if err := tx.Verify(); err != nil {
46              i.t.Config.Context.Log.Debug("Transaction failed verification due to %s,
                    dropping vertex", err)
47              i.t.vtxBlocked.Abandon(vtxID)
48              return
49          }
50      }
51
52      i.t.Config.Context.Log.All("Adding vertex to consensus:\n%s", i.vtx)
53
54      i.t.Consensus.Add(i.vtx)
55
56      p := i.t.Consensus.Parameters()
57      vdrs := i.t.Config.Validators.Sample(p.K) // Validators to sample
58
59      vdrSet := ids.ShortSet{} // Validators to sample repr. as a set
60      for _, vdr := range vdrs {
61          vdrSet.Add(vdr.ID())
62      }
63
64      i.t.RequestID++
65      if numVdrs := len(vdrs); numVdrs == p.K && i.t.polls.Add(i.t.RequestID, vdrSet.
            Len()) {
66          i.t.Config.Sender.PushQuery(vdrSet, i.t.RequestID, vtxID, i.vtx.Bytes())
67      } else if numVdrs < p.K {
68          i.t.Config.Context.Log.Error("Query for %s was dropped due to an
                insufficient number of validators", vtxID)
69      }
70
71      i.t.vtxBlocked.Fulfill(vtxID)
72      for _, tx := range i.vtx.Txs() {
73          i.t.txBlocked.Fulfill(tx.ID())
74      }
75  }
```

Listing 3.20: snow/engine/avalanche/issuer.go

If `i.abandoned` or `i.vtxDeps.Len()!= 0`, it would return immediately. When dependencies are resolved, node would call `i.t.vtxBlocked.Fulfill(vtxID)` to remove the corresponding ID in `i.vtxDeps`. In the end, the vertex would be added into consensus (line 54).

However, this `issuer` could be abandoned on purpose by intentionally triggering `GetFailed`.

```go
func (t *Transitive) GetFailed(vdr ids.ShortID, requestID uint32, vtxID ids.ID) {
    if !t.bootstrapped {
        t.bootstrapper.GetFailed(vdr, requestID, vtxID)
        return
    }

    t.pending.Remove(vtxID)
    t.vtxBlocked.Abandon(vtxID)
    t.vtxReqs.Remove(vtxID)

    if t.vtxReqs.Len() == 0 {
        for _, txID := range t.missingTxs.List() {
            t.txBlocked.Abandon(txID)
        }
        t.missingTxs.Clear()
    }

    // Track performance statistics
    t.numVtxRequests.Set(float64(t.vtxReqs.Len()))
    t.numTxRequests.Set(float64(t.missingTxs.Len()))
    t.numBlockedVtx.Set(float64(t.pending.Len()))
}
```

Listing 3.21: snow/engine/avalanche/transitive.go

Specifically, an attacker could send `Put` message with an invalid vertex with a target vtxID, and the following process would abandon the vertex. Since the vtxID is passed in by user directly, attackers could target any vertex he wants to abandon.

```go
func (b *Blocker) Abandon(id ids.ID) {
     b.init()

     key := id.Key()
     blocking := (*b)[key]
     delete(*b, key)

     for _, pending := range blocking {
      pending.Abandon(id)
     }
}
```

Listing 3.22: snow/consensus/avalanche/topological.go

Corresponding `convincer` would also be affected due to the chain reaction of abandon process. In the end, the node can not respond to a PushQuery message sent by other validators.

Recommendation Enhance the sanity check on the input vertices.

3.8 DoS Vulnerability in the Avalanche Consensus Module - #1

- ID: PVE-008
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `snow/consensus/avalanche/topological.go`
- Category: Coding Practices [7]
- CWE subcategory: CWE-20 [8]

Description

This is a vulnerability in gecko's Avalanche consensus module, which could be exploited by attackers to perform DoS attack against the AVA network.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

A node would send a Put message in response to receiving a Get message for a container the node has access to.

```go
func (t *Transitive) Put(vdr ids.ShortID, requestID uint32, vtxID ids.ID, vtxBytes
    []byte) {
  t.Config.Context.Log.All("Put called for vertexID %s", vtxID)

  if !t.bootstrapped {
      t.bootstrapper.Put(vdr, requestID, vtxID, vtxBytes)
      return
  }

  vtx, err := t.Config.State.ParseVertex(vtxBytes)
  if err != nil {
      t.Config.Context.Log.Warn("ParseVertex failed due to %s for block:\n%s",
      err,
      formatting.DumpBytes{Bytes: vtxBytes})
        t.GetFailed(vdr, requestID, vtxID)
      return
  }
  t.insertFrom(vdr, vtx)
}
```

Listing 3.23: snow/engine/avalanche/transitive.go

`Put` handles an incoming Put request from other validators, it would call `State.ParseVertex` to decode the `Container`

The `ParseVertex` function here simply checks whether the vertex parentIDs and transactions are sorted and unique. If so, the function insertFrom and insert would be called to pass the vertex to consensus engine.

```go
for _, tx := range txs {
    for _, dep := range tx.Dependencies() {
```

```
            depID := dep.ID()
            if !txIDs.Contains(depID) && !t.Consensus.TxIssued(dep) {
                t.missingTxs.Add(depID)
                i.txDeps.Add(depID)
            }
        }
    }

    t.Config.Context.Log.All("Vertex: %s is blocking on %d vertices and %d transactions", vtxID, i.vtxDeps.Len(), i.txDeps.Len())

    t.vtxBlocked.Register(&vtxIssuer{i: i})
    t.txBlocked.Register(&txIssuer{i: i})
```

Listing 3.24: snow/engine/avalanche/transitive.go

As we can see here, two new issuers would be registered in `vtxBlocked` and `txBlocked`. When they are registered, their `Update` function would be called.

```
    func (i *issuer) Update() {
        if i.abandoned || i.issued || i.vtxDeps.Len() != 0 || i.txDeps.Len() != 0 || i.t.Consensus.VertexIssued(i.vtx) {
            return
        }
        i.issued = true

        vtxID := i.vtx.ID()
        i.t.pending.Remove(vtxID)

        for _, tx := range i.vtx.Txs() {
            if err := tx.Verify(); err != nil {
                i.t.Config.Context.Log.Debug("Transaction failed verification due to %s, dropping vertex", err)
                i.t.vtxBlocked.Abandon(vtxID)
                return
            }
        }

        i.t.Config.Context.Log.All("Adding vertex to consensus:\n%s", i.vtx)

        i.t.Consensus.Add(i.vtx)
```

Listing 3.25: snow/engine/avalanche/issuer.go

`i.t.Consensus.Add(i.vtx)` would be called after passing the semantic legitimacy of vtx's transactions.

```
    func (ta *Topological) Add(vtx Vertex) {
        ta.ctx.Log.AssertTrue(vtx != nil, "Attempting to insert nil vertex")

        key := vtx.ID().Key()
        if vtx.Status().Decided() {
            return // Already decided this vertex
```

```go
        } else if _, exists := ta.nodes[key]; exists {
            return // Already inserted this vertex
        }

        for _, tx := range vtx.Txs() {
            if !tx.Status().Decided() {
                // Add the consumers to the conflict graph.
                ta.cg.Add(tx)
            }
        }

        ta.nodes[key] = vtx // Add this vertex to the set of nodes
        ta.numProcessing.Inc()

        ta.update(vtx) // Update the vertex and it's ancestry
    }
```

Listing 3.26: snow/consensus/avalanche/topological.go

Add calls `ta.update` to accept the vertex. Finally, `db.Commit` would store it into database.

```go
360     // Check my parent statuses
361     for _, dep := range deps {
362         if status := dep.Status(); status == choices.Rejected {
363             vtx.Reject() // My parent is rejected, so I should be rejected
364             ta.numRejected.Inc()
365             delete(ta.nodes, vtxKey)
366             ta.numProcessing.Dec()
367
368             ta.preferenceCache[vtxKey] = false
369             ta.virtuousCache[vtxKey] = false
370             return
371         } else if status != choices.Accepted {
372             acceptable = false // My parent isn't accepted, so I can't be
373         }
374     }
```

Listing 3.27: snow/consensus/avalanche/topological.go

However, there is not enough sanity check on the input vertices.

Specifically, an attacker could craft lots of vertices with valid contents, all the sanity checks could be easily bypassed if a crafted vertex has nil `txs` and different `Rejected` parents. In the end, `db.Commit` would store them into database(line 363), which could cause the victim nodes Out-of-Storage.

Furthermore, the victims would pass the vertices to other sampled validators by calling `PushQuery`, and the new victims would repeat this process, resulting in a DDoS attack eventually.

Recommendation Enhance the sanity check on the incoming vertices.

3.9 DoS Vulnerability in the Avalanche Consensus Module - #2

- ID: PVE-009
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `snow/consensus/avalanche/topological.go`
- Category: Coding Practices [7]
- CWE subcategory: CWE-20 [8]

Description

There is a vulnerability in gecko's Avalanche consensus module, which could be exploited by attackers to perform DoS attack against the AVA network.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

A node would send a Put message in response to receiving a Get message for a container the node has access to.

```
func (t *Transitive) Put(vdr ids.ShortID, requestID uint32, vtxID ids.ID, vtxBytes []byte) {
    t.Config.Context.Log.All("Put called for vertexID %s", vtxID)

    if !t.bootstrapped {
        t.bootstrapper.Put(vdr, requestID, vtxID, vtxBytes)
        return
    }

    vtx, err := t.Config.State.ParseVertex(vtxBytes)
    if err != nil {
        t.Config.Context.Log.Warn("ParseVertex failed due to %s for block:\n%s",
        err,
        formatting.DumpBytes{Bytes: vtxBytes})
          t.GetFailed(vdr, requestID, vtxID)
        return
    }
    t.insertFrom(vdr, vtx)
}
```

Listing 3.28: snow/engine/avalanche/transitive.go

`Put` handles an incoming Put request from other validators, it would call `State.ParseVertex` to decode the `Container`

The `ParseVertex` function here simply checks whether the vertex parentIDs and transactions are sorted and unique. If so, the function insertFrom and insert would be called to pass the vertex to consensus engine.

```
for _, tx := range txs {
    for _, dep := range tx.Dependencies() {
```

```go
            depID := dep.ID()
            if !txIDs.Contains(depID) && !t.Consensus.TxIssued(dep) {
                t.missingTxs.Add(depID)
                i.txDeps.Add(depID)
            }
        }
    }

    t.Config.Context.Log.All("Vertex: %s is blocking on %d vertices and %d transactions", vtxID, i.vtxDeps.Len(), i.txDeps.Len())

    t.vtxBlocked.Register(&vtxIssuer{i: i})
    t.txBlocked.Register(&txIssuer{i: i})
```

Listing 3.29: snow/engine/avalanche/transitive.go

As we can see here, two new issuers would be registered in `vtxBlocked` and `txBlocked`. When they are registered, their `Update` function would be called.

```go
func (i *issuer) Update() {
    if i.abandoned || i.issued || i.vtxDeps.Len() != 0 || i.txDeps.Len() != 0 || i.t.Consensus.VertexIssued(i.vtx) {
        return
    }
    i.issued = true

    vtxID := i.vtx.ID()
    i.t.pending.Remove(vtxID)

    for _, tx := range i.vtx.Txs() {
        if err := tx.Verify(); err != nil {
            i.t.Config.Context.Log.Debug("Transaction failed verification due to %s, dropping vertex", err)
            i.t.vtxBlocked.Abandon(vtxID)
            return
        }
    }

    i.t.Config.Context.Log.All("Adding vertex to consensus:\n%s", i.vtx)

    i.t.Consensus.Add(i.vtx)
```

Listing 3.30: snow/engine/avalanche/issuer.go

`i.t.Consensus.Add(i.vtx)` will be called after passing the semantic legitimacy of vtx's transactions.

```go
func (ta *Topological) Add(vtx Vertex) {
    ta.ctx.Log.AssertTrue(vtx != nil, "Attempting to insert nil vertex")

    key := vtx.ID().Key()
    if vtx.Status().Decided() {
        return // Already decided this vertex
    } else if _, exists := ta.nodes[key]; exists {
```

```
            return // Already inserted this vertex
        }

        for _, tx := range vtx.Txs() {
            if !tx.Status().Decided() {
                // Add the consumers to the conflict graph.
                ta.cg.Add(tx)
            }
        }

        ta.nodes[key] = vtx // Add this vertex to the set of nodes
        ta.numProcessing.Inc()

        ta.update(vtx) // Update the vertex and it's ancestry
    }
```

Listing 3.31: snow/consensus/avalanche/topological.go

`Add` calls `ta.update` to accept the vertex. Finally, `db.Commit` will store it into database.

```
412     switch {
413     case acceptable:
414         // I'm acceptable, why not accept?
415         vtx.Accept()
416         ta.numAccepted.Inc()
417         delete(ta.nodes, vtxKey)
418         ta.numProcessing.Dec()
419     case rejectable:
420         // I'm rejectable, why not reject?
421         vtx.Reject()
422         ta.numRejected.Inc()
423         delete(ta.nodes, vtxKey)
424         ta.numProcessing.Dec()
425     }
```

Listing 3.32: snow/consensus/avalanche/topological.go

Although there is a lack of sanity check on the vertices input, to be exact, an attacker could craft lots of vertices with valid contents, `parseVertex` would successfully deserialize them and `vtx.Verify` can be easily bypassed if:

1. a crafted vertex has nil `parentID` and nil `txs` but different `height` will be `Accept()` (line 415).
2. a crafted vertex contains different rejected txs will be `Reject()` (line 421).

In the end, db.Commit will store them into database, which could cause the victim nodes Out-of-Storage.

Furthermore, the victims would pass the vertices to other sampled validators by calling `PushQuery`, and the new victims would repeat this process, resulting in a DDoS attack eventually.

Recommendation Enhance the sanity check on the vertices input.

3.10 DoS Vulnerability in the Avalanche Consensus Module - #3

- ID: PVE-010
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: snow/engine/avalanche/transitive.go
- Category: Bad Coding Practices [7]
- CWE subcategory: CWE-20 [8]

Description

There is a vulnerability in gecko's Avalanche consensus module, which could be exploited by attackers to perform DoS attack against the AVA network.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

A node would send a Put message in response to receiving a Get message for a container the node has access to.

```
func (t *Transitive) Put(vdr ids.ShortID, requestID uint32, vtxID ids.ID, vtxBytes []byte) {
  t.Config.Context.Log.All("Put called for vertexID %s", vtxID)

  if !t.bootstrapped {
    t.bootstrapper.Put(vdr, requestID, vtxID, vtxBytes)
    return
  }

  vtx, err := t.Config.State.ParseVertex(vtxBytes)
  if err != nil {
    t.Config.Context.Log.Warn("ParseVertex failed due to %s for block:\n%s",
      err,
      formatting.DumpBytes{Bytes: vtxBytes})
    t.GetFailed(vdr, requestID, vtxID)
    return
  }
  t.insertFrom(vdr, vtx)
}
```

Listing 3.33: snow/engine/avalanche/transitive.go

`Put` handles an incoming Put request from other validators, it will call `State.ParseVertex` to decode the Container.

```
func (s *Serializer) ParseVertex(b []byte) (avacon.Vertex, error) {
  vtx, err := s.parseVertex(b)
  if err != nil {
```

```
            return nil, err
        }
        if err := vtx.Verify(); err != nil {
            return nil, err
        }
        uVtx := &uniqueVertex{
            serializer: s,
            vtxID:      vtx.ID(),
        }
        if uVtx.Status() == choices.Unknown {
            uVtx.setVertex(vtx)
        }

        s.db.Commit()
        return uVtx, nil
    }
```

Listing 3.34: snow/engine/avalanche/state/serializer.go

```
func (vtx *vertex) Verify() error {
    switch {
    case !ids.IsSortedAndUniqueIDs(vtx.parentIDs):
        return errInvalidParents
    case !isSortedAndUniqueTxs(vtx.txs):
        return errInvalidTxs
    default:
        return nil
    }
}
```

Listing 3.35: snow/engine/avalanche/state/vertex.go

`ParseVertex` calls `parseVertex` to deserialize the vertex, then check the validity of it by vtx.Verify, which will make sure parentIDs and txs are sorted and unique. Finally, db.Commit will store it into database.

Although there is a lack of sanity check on the vertices input, to be exact, an attacker could craft lots of vertices with valid contents, `parseVertex` would successfully deserialize them and vtx.Verify can be easily bypassed if a crafted vertex has unique and sorted parentID / txs (or none of these at all). In the end, db.Commit would store them into database, which could cause the victim nodes Out-of-Storage.

Furthermore, the victims would pass the vertices to other sampled validators by calling `PushQuery` , and the new victims would repeat this process, resulting in a DDoS attack eventually.

Recommendation Add check on vertices with accepted transactions.

3.11 DoS Vulnerability in the Avalanche Consensus Module - #4

- ID: PVE-011
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: snow/engine/avalanche/transitive.go
- Category: Bad Coding Practices [7]
- CWE subcategory: CWE-20 [8]

Description

There is a vulnerability in gecko's Avalanche consensus module, which could be exploited by attackers to perform DoS attack against the AVA network.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

A node would send a Put message in response to receiving a Get message for a container the node has access to.

```go
func (t *Transitive) Put(vdr ids.ShortID, requestID uint32, vtxID ids.ID, vtxBytes []byte) {
  t.Config.Context.Log.All("Put called for vertexID %s", vtxID)

  if !t.bootstrapped {
    t.bootstrapper.Put(vdr, requestID, vtxID, vtxBytes)
    return
  }

  vtx, err := t.Config.State.ParseVertex(vtxBytes)
  if err != nil {
    t.Config.Context.Log.Warn("ParseVertex failed due to %s for block:\n%s",
      err,
      formatting.DumpBytes{Bytes: vtxBytes})
    t.GetFailed(vdr, requestID, vtxID)
    return
  }
  t.insertFrom(vdr, vtx)
}
```

Listing 3.36: snow/engine/avalanche/transitive.go

`Put` handles incoming Put messages, it would use State.`ParseVertex` to decode the Container

`ParseVertex` simply checks whether the vertex parentIDs and transactions are sorted and unique. If so, `insertFrom` and `insert` will be called to pass the vertex to consensus engine.

```go
func (t *Transitive) insertFrom(vdr ids.ShortID, vtx avalanche.Vertex) bool {
```

```go
	issued := true
	vts := []avalanche.Vertex{vtx}
	for len(vts) > 0 {
		vtx := vts[0]
		vts = vts[1:]

		if t.Consensus.VertexIssued(vtx) {
			continue
		}
		if t.pending.Contains(vtx.ID()) {
			issued = false
			continue
		}

		for _, parent := range vtx.Parents() {
			if !parent.Status().Fetched() {
				t.sendRequest(vdr, parent.ID())
				issued = false
			} else {
				vts = append(vts, parent)
			}
		}

		t.insert(vtx)
	}
	return issued
}
```

Listing 3.37: snow/engine/avalanche/transitive.go

```go
func (t *Transitive) insert(vtx avalanche.Vertex) {
	vtxID := vtx.ID()

	t.pending.Add(vtxID)
	t.vtxReqs.Remove(vtxID)

	i := &issuer{
		t:   t,
		vtx: vtx,
	}

	for _, parent := range vtx.Parents() {
		if !t.Consensus.VertexIssued(parent) {
			i.vtxDeps.Add(parent.ID())
		}
	}

	txs := vtx.Txs()

	txIDs := ids.Set{}
	for _, tx := range txs {
		txIDs.Add(tx.ID())
	}
```

```go

  for _, tx := range txs {
    for _, dep := range tx.Dependencies() {
      depID := dep.ID()
      if !txIDs.Contains(depID) && !t.Consensus.TxIssued(dep) {
        t.missingTxs.Add(depID)
        i.txDeps.Add(depID)
      }
    }
  }

  t.Config.Context.Log.All("Vertex: %s is blocking on %d vertices and %d transactions",
      vtxID, i.vtxDeps.Len(), i.txDeps.Len())

  t.vtxBlocked.Register(&vtxIssuer{i: i})
  t.txBlocked.Register(&txIssuer{i: i})

  if t.vtxReqs.Len() == 0 {
    for _, txID := range t.missingTxs.List() {
      t.txBlocked.Abandon(txID)
    }
    t.missingTxs.Clear()
  }

  // Track performance statistics
  t.numVtxRequests.Set(float64(t.vtxReqs.Len()))
  t.numTxRequests.Set(float64(t.missingTxs.Len()))
  t.numBlockedVtx.Set(float64(t.pending.Len()))
}
```

Listing 3.38: snow/engine/avalanche/transitive.go

As we can see here, if the vertex's parents haven't been fectched, it would call `sendRequest` to get the parent vertex. And two new issuers would be registered in vtxBlocked and txBlocked. When registered, `Update` will be called.

```go
func (i *issuer) Update() {
  if i.abandoned || i.issued || i.vtxDeps.Len() != 0 || i.txDeps.Len() != 0 || i.t.
      Consensus.VertexIssued(i.vtx) {
    return
  }
  i.issued = true

  vtxID := i.vtx.ID()
  i.t.pending.Remove(vtxID)

  for _, tx := range i.vtx.Txs() {
    if err := tx.Verify(); err != nil {
      i.t.Config.Context.Log.Debug("Transaction failed verification due to %s, dropping
          vertex", err)
      i.t.vtxBlocked.Abandon(vtxID)
      return
```

```
		}
	}

	i.t.Config.Context.Log.All("Adding vertex to consensus:\n%s", i.vtx)

	i.t.Consensus.Add(i.vtx)

	p := i.t.Consensus.Parameters()
	vdrs := i.t.Config.Validators.Sample(p.K) // Validators to sample

	vdrSet := ids.ShortSet{} // Validators to sample repr. as a set
	for _, vdr := range vdrs {
		vdrSet.Add(vdr.ID())
	}

	i.t.RequestID++
	if numVdrs := len(vdrs); numVdrs == p.K && i.t.polls.Add(i.t.RequestID, vdrSet.Len()) {
		i.t.Config.Sender.PushQuery(vdrSet, i.t.RequestID, vtxID, i.vtx.Bytes())
	} else if numVdrs < p.K {
		i.t.Config.Context.Log.Error("Query for %s was dropped due to an insufficient number of validators", vtxID)
	}

	i.t.vtxBlocked.Fulfill(vtxID)
	for _, tx := range i.vtx.Txs() {
		i.t.txBlocked.Fulfill(tx.ID())
	}
}
```

Listing 3.39: snow/engine/avalanche/issuer.go

If `sendRequest` timed out, `GetFailed` will clear the registered issuers.

```
func (t *Transitive) GetFailed(vdr ids.ShortID, requestID uint32, vtxID ids.ID) {
	if !t.bootstrapped {
		t.bootstrapper.GetFailed(vdr, requestID, vtxID)
		return
	}

	t.pending.Remove(vtxID)
	t.vtxBlocked.Abandon(vtxID)
	t.vtxReqs.Remove(vtxID)

	if t.vtxReqs.Len() == 0 {
		for _, txID := range t.missingTxs.List() {
			t.txBlocked.Abandon(txID)
		}
		t.missingTxs.Clear()
	}

	// Track performance statistics
	t.numVtxRequests.Set(float64(t.vtxReqs.Len()))
```

```
    t.numTxRequests.Set(float64(t.missingTxs.Len()))
    t.numBlockedVtx.Set(float64(t.pending.Len()))
}
```

Listing 3.40: snow/engine/avalanche/transitive.go

However, if the length of vtxReqs is not equal to zero, the corresponding issuers in txBlocked won't be deleted and missingTxs won't be cleared.

Specifically, an attacker could send lots of vertices with non-existent vertex parents to flood the txBlocked and missingTxs. Eventually, this would cause the node Out-of-Memory.

Recommendation Delete the corresponding txBlocked and missingTxs after timeout.

3.12 Missing Sanity Check in the Bootstrapping Process

- ID: PVE-012
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `snow/engine/avalanche/bootstrapper.go`
- Category: Input Validation Issues [11]
- CWE subcategory: CWE-349 [12]

Description

This is a vulnerability in the consensus module of gecko, which could be exploited by attackers to compromise the AVA network consensus.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

A node would send a Put message in response to receiving a Get message for a container the node has access to.

```
func (b *bootstrapper) Put(vdr ids.ShortID, requestID uint32, vtxID ids.ID, vtxBytes []byte) {
    b.BootstrapConfig.Context.Log.All("Put called for vertexID %s", vtxID)

    if !b.pending.Contains(vtxID) {
        return
    }

    vtx, err := b.State.ParseVertex(vtxBytes)
    if err != nil {
        b.BootstrapConfig.Context.Log.Warn("ParseVertex failed due to %s for block:\n%s",
            err,
            formatting.DumpBytes{Bytes: vtxBytes})
        b.GetFailed(vdr, requestID, vtxID)
```

```
        return
    }

    b.addVertex(vtx)
}
```

Listing 3.41: snow/engine/avalanche/bootstrapper.go

`ParseVertex` would try to extract the vertex from vtxBytes, and the vtx.status would be set to Processing if extracted successfully. After that, `addVertex` would create vertexJob / txJob for the received vertex and transactions, and push them to VtxBlocked and TxBlocked as long as the vtxID is in pending.

At last, `executeAll` would execute the tx / vertex job, to accept or reject them.

```
func (b *bootstrapper) executeAll(jobs *queue.Jobs, numBlocked prometheus.Gauge) {
    for job, err := jobs.Pop(); err == nil; job, err = jobs.Pop() {
        numBlocked.Dec()
        if err := jobs.Execute(job); err != nil {
            b.BootstrapConfig.Context.Log.Warn("Error executing: %s", err)
        }
    }
}
```

Listing 3.42: snow/engine/avalanche/bootstrapper.go

`jobs.Execute` execute the job's Execute, for example, txJob.Execute for the transactions.

```
44 func (t *txJob) Execute() {
45     if t.MissingDependencies().Len() != 0 {
46         t.numDropped.Inc()
47         return
48     }
49
50     switch t.tx.Status() {
51     case choices.Unknown, choices.Rejected:
52         t.numDropped.Inc()
53     case choices.Processing:
54         t.tx.Accept()
55         t.numAccepted.Inc()
56     }
57 }
```

Listing 3.43: snow/engine/avalanche/tx_job.go

However, in the case of choices.Processing, transactions could be accepted without any sanity checks (line 54), e.g., tx.Verify for the validity of the transaction.

Specifically, an attacker could send Put messages with a vtxID in pending and an illegal vertex to other nodes, and the illegal transaction / vertex would be accepted unconditionally since there is no sanity checks at all.

Recommendation Add check whether blkID corresponds to blkBytes in Put function.

3.13 Missing Sanity Check in the Snowman Consensus Process - #1

- ID: PVE-013
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: snow/engine/snowman/polls.go
- Category: Coding Practices [7]
- CWE subcategory: CWE-20 [8]

Description

This is a vulnerability in gecko's Snowman consensus module, which could be exploited by attackers to compromise the AVA network consensus.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

A node would send a Chits message in response to receiving a PullQuery or PushQuery message for a container the node has added to consensus.

```go
// Chits implements the Engine interface
func (t *Transitive) Chits(vdr ids.ShortID, requestID uint32, votes ids.Set) {
    // Since this is snowman, there should only be one ID in the vote set
    if votes.Len() != 1 {
        t.Config.Context.Log.Warn("Chits was called with the wrong number of votes %
            d. RequestID: %d", votes.Len(), requestID)
        t.QueryFailed(vdr, requestID)
        return
    }
    vote := votes.List()[0]

    t.Config.Context.Log.All("Chit was called. RequestID: %v. Vote: %s", requestID,
        vote)

    v := &voter{
        t:         t,
        vdr:       vdr,
        requestID: requestID,
        response:  vote,
    }

    if !t.reinsertFrom(vdr, vote) {
        v.deps.Add(vote)
    }

    t.blocked.Register(v)
}
```

Listing 3.44: snow/engine/snowman/transitive.go

```
24  func (v *voter) Update() {
25      if v.deps.Len() != 0 {
26          return
27      }
28
29      results := ids.Bag{}
30      finished := false
31      if v.response.IsZero() {
32          results, finished = v.t.polls.CancelVote(v.requestID, v.vdr)
33      } else {
34          results, finished = v.t.polls.Vote(v.requestID, v.vdr, v.response)
35      }
36
37      if !finished {
38          return
39      }
40
41      v.t.Config.Context.Log.All("Finishing poll [%d] with:\n%s", v.requestID, results)
42      v.t.Consensus.RecordPoll(results)
43
44      v.t.Config.VM.SetPreference(v.t.Consensus.Preference())
45
46      if v.t.Consensus.Finalized() {
47          v.t.Config.Context.Log.All("Snowman engine can quiesce")
48          return
49      }
50
51      v.t.Config.Context.Log.All("Snowman engine can't quiesce")
52
53      if len(v.t.polls.m) == 0 {
54          v.t.repoll()
55      }
56  }
```

Listing 3.45: snow/engine/snowman/voter.go

When the vote is registered, `Update` would be called and executes `polls.Vote` for voting (line 34), or cancel the vote (line 32).

```
func (p *polls) Vote(requestID uint32, vdr ids.ShortID, vote ids.ID) (ids.Bag, bool) {
    p.log.All("[polls.Vote] Vote: requestID: %d. validatorID: %s. Vote: %s", requestID, vdr, vote)
    poll, exists := p.m[requestID]
    if !exists {
        return ids.Bag{}, false
    }
    poll.Vote(vote)
    if poll.Finished() {
        delete(p.m, requestID)
        p.numPolls.Set(float64(len(p.m))) // Tracks performance statistics
        return poll.votes, true
```

```
        }
        p.m[requestID] = poll
        return ids.Bag{}, false
    }

    // CancelVote registers the connections failure to respond to a query for [id].
    func (p *polls) CancelVote(requestID uint32, vdr ids.ShortID) (ids.Bag, bool) {
        p.log.All("CancelVote received. requestID: %d. validatorID: %s. Vote: %s",
            requestID, vdr)
        poll, exists := p.m[requestID]
        if !exists {
            return ids.Bag{}, false
        }

        poll.CancelVote()
        if poll.Finished() {
            delete(p.m, requestID)
            p.numPolls.Set(float64(len(p.m))) // Tracks performance statistics
            return poll.votes, true
        }
        p.m[requestID] = poll
        return ids.Bag{}, false
    }
```

Listing 3.46: snow/engine/snowman/polls.go

However, neither of these two has any sanity checks on whether the validator has already voted. The `vdr` argument which is supposed to be used for checking the duplicate votes is never used here.

Specifically, an attacker could vote many times to finish the poll before the victim gets votes from other validators. Since `requestId` is decoded from the message directly, attackers could target any `requestId` it wants to interrupt.

Recommendation Add a sanity check to forbid duplicate votes.

3.14 Missing Sanity Check in the Snowman Consensus Process - #2

- ID: PVE-014
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `snow/engine/snowman/transitive.go`
- Category: Input Validation Issues [11]
- CWE subcategory: CWE-349 [12]

Description

There is a vulnerability in gecko's Snowman consensus module, which could be exploited by attackers to compromise the AVA network consensus.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

A node would send a Put message in response to receiving a Get message for a container the node has access to.

```go
func (t *Transitive) Put(vdr ids.ShortID, requestID uint32, blkID ids.ID, blkBytes []byte) {
  blk, err := t.Config.VM.ParseBlock(blkBytes)
  if err != nil {
    t.Config.Context.Log.Warn("ParseBlock failed due to %s for block:\n%s",
      err,
      formatting.DumpBytes{Bytes: blkBytes})
    t.GetFailed(vdr, requestID, blkID)
    return
  }

  t.insertFrom(vdr, blk)
}
```

Listing 3.47: snow/engine/snowman/transitive.go

`Put` handles incoming Put messages, it would call `VM.ParseBlock` to decode the Container.

`insertFrom` and `insert` would be called if the block can be unmarshalled. If the block's parent isn't fetched yet, the node would try `sendRequest(vdr, parentID)` to get it back.

```go
func (t *Transitive) insertFrom(vdr ids.ShortID, blk snowman.Block) bool {
  blkID := blk.ID()
  for !t.Consensus.Issued(blk) && !t.pending.Contains(blkID) {
    t.insert(blk)

    parent := blk.Parent()
    parentID := parent.ID()
    if parentStatus := parent.Status(); !parentStatus.Fetched() {
      t.sendRequest(vdr, parentID)
      return false
    }

    blk = parent
    blkID = parentID
  }
  return !t.pending.Contains(blkID)
}
```

Listing 3.48: snow/engine/snowman/transitive.go

```go
func (t *Transitive) insert(blk snowman.Block) {
```

```
  blkID := blk.ID()

  t.pending.Add(blkID)
  t.blkReqs.Remove(blkID)

  i := &issuer{
    t:   t,
    blk: blk,
  }

  if parent := blk.Parent(); !t.Consensus.Issued(parent) {
    parentID := parent.ID()
    t.Config.Context.Log.All("Block waiting for parent %s", parentID)
    i.deps.Add(parentID)
  }

  t.blocked.Register(i)

  // Tracks performance statistics
  t.numBlkRequests.Set(float64(t.blkReqs.Len()))
  t.numBlockedBlk.Set(float64(t.pending.Len()))
}
```

Listing 3.49: snow/engine/snowman/transitive.go

As we can see, the new issuer would be registered to blocked. If they have dependencies which haven't been issued, t.blocked would record that issuer to its corresponding depID keys in the map. Finally, `Update` would be called.

```
func (b *Blocker) Register(pending Blockable) {
  b.init()

  for _, pendingID := range pending.Dependencies().List() {
    key := pendingID.Key()
    (*b)[key] = append((*b)[key], pending)
  }

  pending.Update()
}
```

Listing 3.50: snow/events/blocker.go

```
func (i *issuer) Update() {
  if i.abandoned || i.deps.Len() != 0 {
    return
  }

  i.t.deliver(i.blk)
}
```

Listing 3.51: snow/engine/snowman/issuer.go

If i.abandoned or i.deps.Len() != 0, it would return immediately. When dependencies are resolved, deliver would be called to add the block into consensus and send it out (line 40).

However, this issuer could be abandoned on purpose by intentionally triggering `GetFailed`.

Specifically, an attacker could send Put message with an invalid block which will fail `VM.ParseBlock` and the following process would abandon the block. Since the block ID is passed in by user directly, attacker can target any block he wants to abandon.

```
35 func (b *Blocker) Abandon(id ids.ID) {
36   b.init()
37 
38   key := id.Key()
39   blocking := (*b)[key]
40   delete(*b, key)
41 
42   for _, pending := range blocking {
43     pending.Abandon(id)
44   }
45 }
```

Listing 3.52: snow/events/blocker.go

Corresponding `convincer` would also be affected due to the chain reaction of abandon process. In the end, the node can not response to a `PushQuery` message sent by other validators.

Recommendation Add check whether blkID corresponds to blkBytes in Put function.

3.15 Missing Sanity Check in the RPC Module

- ID: PVE-015
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `api/admin/service.go`
- Category: Input Validation Issues [11]
- CWE subcategory: CWE-349 [12]

Description

This is a vulnerability in the RPC module of gecko, which could be exploited by attackers to overwrite arbitrary files on the node server if the node is running with root privilege.

Ava node provides RPC service for users to interact with the node. The `admin.startCPUProfiler` and `admin.memoryProfile` APIs are used to dump the profile information into the specified file. The node handles the rpc call with the function `StartCPUProfiler`.

```
func (service *Admin) StartCPUProfiler(r *http.Request, args *StartCPUProfilerArgs,
    reply *StartCPUProfilerReply) error {
  service.log.Debug("Admin: StartCPUProfiler called with %s", args.Filename)
```

```
  reply.Success = true
  return service.performance.StartCPUProfiler(args.Filename)
}
```

Listing 3.53: api/admin/service.go

It would get the filename from the arguments and write the profile information into that file.

```
func (p *Performance) StartCPUProfiler(filename string) error {
  if p.cpuProfileFile != nil {
    return errCPUProfilerRunning
  }

  file, err := os.Create(filename)
  if err != nil {
    return err
  }
  if err := pprof.StartCPUProfile(file); err != nil {
    file.Close()
    return err
  }
  runtime.SetMutexProfileFraction(1)

  p.cpuProfileFile = file
  return nil
}
```

Listing 3.54: api/admin/performance.go

However, there is no sanity checks on the filename passed in, nor any privilege requirements on the rpc call.

Specifically, the attacker could send the target file path and file name to the rpc call (like "../../etc/passwd") and the file would be overwritten by the node depends on the node process's privilege.

Recommendation Add check on input file path to prevent traversal attack.

3.16 DoS Vulnerability in the RPC Module

- ID: PVE-016
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `vms/avm/vm.go`
- Category: Coding Practices [7]
- CWE subcategory: CWE-20 [8]

Description

There is a potential performance issue in the the X-Chain API of gecko, which could be exploited by attackers to perform a DoS attack against an AVA node.

```
func (vm *VM) GetUTXOs(addrs ids.Set) ([]*UTXO, error) {
  utxoIDs := ids.Set{}
  for _, addr := range addrs.List() {
    utxos, _ := vm.state.Funds(addr)
    utxoIDs.Add(utxos...)
  }

  utxos := []*UTXO{}
  for _, utxoID := range utxoIDs.List() {
    utxo, err := vm.state.UTXO(utxoID)
    if err != nil {
      return nil, err
    }
    utxos = append(utxos, utxo)
  }
  return utxos, nil
}
```

Listing 3.55: vms/avm/vm.go

`GetUTXOs` returns the utxos that at least one of the provided addresses is referenced in. However, these UTXOs are stored in a single list for each account, and there is no limitation on the length of the passed in addr. Technically, attackers could send requests to this API with a series of addresses which may own numerous utxos, to exhaust the resources of the victim node.

Recommendation Limit the length of the input addr.

3.17 DoS Vulnerability in the P2P Module

- ID: PVE-017
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `networking/handshake_handlers.go`
- Category: Coding Practices [7]
- CWE subcategory: CWE-20 [8]

Description

This is a vulnerability in the P2P module of gecko, which could be exploited by attackers to perform DoS attack against the AVA network.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

```go
    net.RegConnHandler(salticidae.MsgNetworkConnCallback(C.checkPeerCertificate), nil)
    peerNet.RegPeerHandler(salticidae.PeerNetworkPeerCallback(C.peerHandler), nil)
    peerNet.RegUnknownPeerHandler(salticidae.PeerNetworkUnknownPeerCallback(C.unknownPeerHandler), nil)
    net.RegHandler(Ping, salticidae.MsgNetworkMsgCallback(C.ping), nil)
    net.RegHandler(Pong, salticidae.MsgNetworkMsgCallback(C.pong), nil)
    net.RegHandler(GetVersion, salticidae.MsgNetworkMsgCallback(C.getVersion), nil)
    net.RegHandler(Version, salticidae.MsgNetworkMsgCallback(C.version), nil)
    net.RegHandler(GetPeerList, salticidae.MsgNetworkMsgCallback(C.getPeerList), nil)
    net.RegHandler(PeerList, salticidae.MsgNetworkMsgCallback(C.peerList), nil)
```

Listing 3.56: networking/handshake_handlers.go

During node setup process, `Initialize` registers many handlers for different kind of messages.

```go
func peerList(_msg *C.struct_msg_t, _conn *C.struct_msgnetwork_conn_t, _ unsafe.Pointer) {
    HandshakeNet.numPeerlistReceived.Inc()

    msg := salticidae.MsgFromC(salticidae.CMsg(_msg))
    build := Builder{}
    pMsg, err := build.Parse(PeerList, msg.GetPayloadByMove())
    if err != nil {
        HandshakeNet.log.Warn("Failed to parse PeerList message due to %s", err)
        // TODO: What should we do here?
        return
    }

    ips := pMsg.Get(Peers).([]utils.IPDesc)
    cErr := salticidae.NewError()
    for _, ip := range ips {
        HandshakeNet.log.All("Trying to adding peer %s", ip)
        addr := salticidae.NewNetAddrFromIPPortString(ip.String(), false, &cErr)
        if cErr.GetCode() == 0 && !HandshakeNet.myAddr.IsEq(addr) { // Make sure not to connect to myself
            ip := toIPDesc(addr)

            if !HandshakeNet.pending.ContainsIP(addr) && !HandshakeNet.connections.ContainsIP(addr) {
                HandshakeNet.log.Debug("Adding peer %s", ip)
                HandshakeNet.net.AddPeer(addr)
            }
        }
        addr.Free()
    }
}
```

Listing 3.57: networking/handshake_handlers.go

`peerlist` is registered to handle peerList message. If the addr is not in `pending` or `connections`, it will call `AddPeer` to store the new peer.

```go
func (self PeerNetwork) AddPeer(peer PeerID) int32 {
```

```
        return int32(C.peernetwork_add_peer(self.inner, peer.inner))
    }
```

Listing 3.58: network.go

The new peer would be added to `known_peers` which is a `std::unordered_map`.

However, there is no limitation nor any sanity checks on the passed in peers in the `peerlist` message.

Specifically, the attacker could flood the target with peerlist messages containing different peer addresses which could cause the victim nodes Out-of-Memory.

The same problem exists in `unknownPeerHandler`.

```
func unknownPeerHandler(_addr *C.netaddr_t, _cert *C.x509_t, _ unsafe.Pointer) {
        addr := salticidae.NetAddrFromC(salticidae.CNetAddr(_addr))
        ip := toIPDesc(addr)
        HandshakeNet.log.Info("Adding peer %s", ip)
        HandshakeNet.net.AddPeer(addr)
}
```

Listing 3.59: networking/handshake_handlers.go

In C++ library, if the `ping_handler` receives a ping message with unknown claimed address, it would call `unknown_peer_cb`.

`unknown_peer_cb` would call `unknownPeerHandler` to add the unknown peer to `HandshakeNet.net`. Since the addr is decoded from the ping message directly(msg.claimed_addr), the attacker could use a different addresses everytime.

Specifically, the attacker could flood the target with C++ level ping messages(MsgPing) containing different claimed addresses which could cause the victim nodes Out-of-Memory.

Recommendation Use LRU cache or add a length limitation on known_peers.

3.18 DoS Vulnerability in the VM Module - #5

- ID: PVE-018
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `coreth/consensus/dummy/consensus.go`
- Category: Coding Practices [7]
- CWE subcategory: CWE-20 [8]

Description

This is a vulnerability in the EVM module of gecko, which could be exploited by attackers to perform DoS attack against the AVA network.

AVA network defines the core communication format between AVA nodes. It uses the primitive serialization format for payload packing and Salticidae's message format.

A node would send a Put message in response to receiving a Get message for a container the node has access to.

```
func (t *Transitive) Put(vdr ids.ShortID, requestID uint32, blkID ids.ID, blkBytes []byte) {
    blk, err := t.Config.VM.ParseBlock(blkBytes)
    if err != nil {
        t.Config.Context.Log.Warn("ParseBlock failed due to %s for block:\n%s",
            err,
            formatting.DumpBytes{Bytes: blkBytes})
        t.GetFailed(vdr, requestID, blkID)
        return
    }

    t.insertFrom(vdr, blk)
}
```

Listing 3.60: snow/engine/snowman/transitive.go

`Put` handles an incoming Put request from other validators, it would call `VM.ParseBlock` to decode the `Container`.

```
// ParseBlock implements the snowman.ChainVM interface
func (vm *VM) ParseBlock(b []byte) (snowman.Block, error) {
    vm.metalock.Lock()
    defer vm.metalock.Unlock()

    ethBlock := new(types.Block)
    if err := rlp.DecodeBytes(b, ethBlock); err != nil {
        return nil, err
    }
    block := &Block{
        id:       ids.NewID(ethBlock.Hash()),
        ethBlock: ethBlock,
        vm:       vm,
    }
    vm.blockCache.Put(block.ID(), block)
    return block, nil
}
```

Listing 3.61: vms/evm/vm.go

`ParseBlock` calls `DecodeBytes` to decode the raw bytes into `ethBlock` and then assigns it's hash to `Block.id`. Next, snowman checks the block's validity by calling `blk.Verify()`.

```
func (t *Transitive) deliver(blk snowman.Block) {
    if t.Consensus.Issued(blk) {
        return
    }

```

```
        blkID := blk.ID()
        t.pending.Remove(blkID)

        if err := blk.Verify(); err != nil {
            t.Config.Context.Log.Debug("Block failed verification due to %s, dropping
                block", err)
            t.blocked.Abandon(blkID)
            t.numBlockedBlk.Set(float64(t.pending.Len())) // Tracks performance
                statistics
            return
        }

        t.Config.Context.Log.All("Adding block to consensus: %s", blkID)

        t.Consensus.Add(blk)
        t.pushSample(blk)
```

Listing 3.62: snow/engine/snowman/transitive.go

```
    // Verify implements the snowman.Block interface
    func (b *Block) Verify() error {
        _, err := b.vm.chain.InsertChain([]*types.Block{b.ethBlock})
        return err
    }
```

Listing 3.63: vms/evm/block.go

Finally, `InsertChain` calls `verifyHeaderWorker` to confirm the new block's header is valid.

```
    // modified from consensus.go
    func (self *DummyEngine) verifyHeader(chain consensus.ChainReader, header, parent *
        types.Header, uncle bool, seal bool) error {
        // Ensure that the header's extra-data section is of a reasonable size
        if uint64(len(header.Extra)) > myparams.MaximumExtraDataSize {
            return fmt.Errorf("extra-data too long: %d > %d", len(header.Extra),
                myparams.MaximumExtraDataSize)
        }
        // Verify the header's timestamp
        if !uncle {
            if header.Time > uint64(time.Now().Add(allowedFutureBlockTime).Unix()) {
                return consensus.ErrFutureBlock
            }
        }
        //if header.Time <= parent.Time {
        if header.Time < parent.Time {
            return errZeroBlockTime
        }
        // Verify that the gas limit is <= 2^63-1
        cap := uint64(0x7fffffffffffffff)
        if header.GasLimit > cap {
            return fmt.Errorf("invalid gasLimit: have %v, max %v", header.GasLimit, cap)
        }
        // Verify that the gasUsed is <= gasLimit
```

```go
        if header.GasUsed > header.GasLimit {
            return fmt.Errorf("invalid gasUsed: have %d, gasLimit %d", header.GasUsed,
                header.GasLimit)
        }

        // Verify that the gas limit remains within allowed bounds
        diff := int64(parent.GasLimit) - int64(header.GasLimit)
        if diff < 0 {
            diff *= -1
        }
        limit := parent.GasLimit / params.GasLimitBoundDivisor

        if uint64(diff) >= limit || header.GasLimit < params.MinGasLimit {
            return fmt.Errorf("invalid gas limit: have %d, want %d += %d", header.
                GasLimit, parent.GasLimit, limit)
        }
        // Verify that the block number is parent's +1
        if diff := new(big.Int).Sub(header.Number, parent.Number); diff.Cmp(big.NewInt
            (1)) != 0 {
            return consensus.ErrInvalidNumber
        }
        // Verify the engine specific seal securing the block
        if seal {
            if err := self.VerifySeal(chain, header); err != nil {
                return err
            }
        }
        return nil
    }

    func (self *DummyEngine) verifyHeaderWorker(chain consensus.ChainReader, headers []*
        types.Header, seals []bool, index int) error {
        var parent *types.Header
        if index == 0 {
            parent = chain.GetHeader(headers[0].ParentHash, headers[0].Number.Uint64()
                -1)
        } else if headers[index-1].Hash() == headers[index].ParentHash {
            parent = headers[index-1]
        }
        if parent == nil {
            return consensus.ErrUnknownAncestor
        }
        if chain.GetHeader(headers[index].Hash(), headers[index].Number.Uint64()) != nil
         {
            return nil // known block
        }
        return self.verifyHeader(chain, headers[index], parent, false, seals[index])
    }
```

Listing 3.64: coreth/consensus/dummy/consensus.go

However, the implementation of Ethereum Virtual Machine in Avalanche is different from the original PoW one, it doesn't need to calculate the `Difficulty` in the header, so `VerifySeal`(line 98) can be ignored. On the other hand, since it's not base on PoW, the massive calculation of `Header.nonce` against the `MixDigest` is omitted.

```
func (self *DummyEngine) VerifySeal(chain consensus.ChainReader, header *types.
    Header) error {
    return nil
}

func (self *DummyEngine) Prepare(chain consensus.ChainReader, header *types.Header)
    error {
  header.Difficulty = big.NewInt(1)
  return nil
}
```

Listing 3.65: coreth/consensus/dummy/consensus.go

Specifically, an attacker could craft lots of valid blocks with nil txs, i.e. empty blocks and send them out.

Since these blocks are valid, they would be added into consensus, which can be used for flood attacks. On the other hand, attackers could also set `header.Time` to the maximum extent (15 * time.Second), so the next block would not be accepted until that time(lines 62 71), which could significantly decrease the performance.

So technically, attackers could send out lots of valid `empty block` within 15 seconds `header.Time` to paralyze the ETH in the AVA network.

Recommendation Add a sanity check on `header.Time`.

4 | Conclusion

In this security audit, we have analyzed the Avalanche Blockchain. During the first phase of our audit, we studied the source code and ran our in-house analyzing tools through the codebase, including areas such as P2P networking, consensus algorithm, and transaction model, etc. A list of potential issues were found, and some of them involve unusual interactions among multiple modules, therefore we developed test cases to reproduce and verify each of them. After further analysis and internal discussion, we determined that 18 issues need to be brought up and paid more attention to, which are reported in Sections 2 and 3.

Our impression through this audit is that the Avalanche Blockchain software is neatly organized and elegantly implemented and those identified issues are promptly confirmed and fixed. We'd like to commend AVA Labs for a well-done software project, and for quickly fixing issues found during the audit process. Also, as expressed in Section 1.4, we appreciate any constructive feedback or suggestions about this report.

References

[1] AVA Labs. AVA Blockchain. https://github.com/ava-labs/.

[2] AVA Labs. AVA Labs. https://avalabs.org.

[3] PeckShield. PeckShield Inc. https://www.peckshield.com.

[4] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[5] Lcamtuf. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[6] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[8] MITRE. CWE CATEGORY: Improper Input Validation. https://cwe.mitre.org/data/definitions/20.html.

[9] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[10] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

[11] MITRE. CWE-1215: Input Validation Issues. https://cwe.mitre.org/data/definitions/1215.html.

[12] MITRE. CWE-349: Acceptance of Extraneous Untrusted Data With Trusted Data. https://cwe.mitre.org/data/definitions/349.html.