# Ava Labs AvalancheGo

## Security Assessment

**August 8, 2025**

*Prepared for:*
**Martin Prado**
Ava Labs

*Prepared by:* **Bo Henderson, David Pokora, Kevin Valerio, Nicolas Donboly, Benjamin Samuels, and Jay Little**

# Table of Contents


**Table of Contents** 1
**Project Summary** 2
**Executive Summary** 3
**Project Goals** 5
**Project Targets** 6
**Project Coverage** 7
**Summary of Findings** 10
**Detailed Findings** 11
1. Unbounded recursion in codec allows stack overflow via deeply nested structs 11
2. Lack of lower bound on range proof request bytes limit enables denial of service 14
3. LevelDB prone to panic from poor construction 16
4. TOCTOU in the perms package Create method 18
5. API server does not limit large request body sizes 20
**A. Vulnerability Categories** 21
**B. AI Usage** 23
Overview 23
Approaches 23
1. Onboarding through AI-powered analysis 23
2. Identifying high-value areas of concern 24
3. Denial-of-service attack vector evaluation 25
4. Automated fuzzing 25
5. Automated auditing using a judge advocate model 27
6. Vulnerability pattern matching using historical bug datasets 27
Challenges 29
Results Summary 30
Future Considerations 31
**C. AI Agent Denial of Service Attempts** 33
**About Trail of Bits** 37
**Notices and Remarks** 38

# Project Summary

## Contact Information

The following project manager was associated with this project:

**Samuel Greenup**, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

**Benjamin Samuels**, Engineering Director, Blockchain
benjamin.samuels@trailofbits.com

The following consultants were associated with this project:

**Bo Henderson**, Consultant
bo.henderson@trailofbits.com

**David Pokora**, Consultant
david.pokora@trailofbits.com

**Nicolas Donboly**, Consultant
nicolas.donboly@trailofbits.com

**Kevin Valerio**, Consultant
kevin.valerio@trailofbits.com

**Jay Little**, Consultant
jay@trailofbits.com

**Benjamin Samuels**, Consultant
benjamin.samuels@trailofbits.com

## Project Timeline

The significant events and milestones of the project are listed below.

| Date | Event |
|---|---|
| **June 6, 2025** | Pre-project kickoff call |
| **June 13, 2025** | Status update meeting #1 |
| **June 20, 2025** | Status update meeting #2 |
| **July 7, 2025** | Status update meeting #3 |
| **July 14, 2025** | Status update meeting #4 |
| **July 21, 2025** | Delivery of report draft |
| **July 21, 2025** | Report readout meeting |
| **August 8, 2025** | Delivery of final comprehensive report |

# Executive Summary

## Engagement Overview

Ava Labs engaged Trail of Bits to conduct a comprehensive AI-augmented security assessment of the AvalancheGo codebase, the Go node implementation for the Avalanche network. This engagement was specifically designed to evaluate both the effectiveness of AI as an offensive security tool and the codebase's resilience against AI-driven attacks.

A team of six consultants conducted the review from June 9 to July 18, 2025, for a total of 10 engineer-weeks of effort. Our testing efforts focused on leveraging AI to systematically identify vulnerabilities, generate attack vectors, and simulate how modern adversaries might use AI to target the AvalancheGo implementation. With full access to source code and documentation, we performed static and dynamic testing using a combination of traditional security methods enhanced with cutting-edge AI capabilities.

## Observations and Impact

Our assessment identified five informational-severity issues that highlight areas for improvement but do not pose immediate security risks. The absence of any high- or medium-severity vulnerabilities reflects the Ava Labs team's commitment to strong security practices and the maturity gained from operating long term in adversarial environments.

Two vulnerabilities were initially discovered through manual review: an unbounded recursion vulnerability in the codec that could enable stack overflow attacks (TOB-AVAX-1) and a denial-of-service vector in MerkleDB range proof requests with small byte limits (TOB-AVAX-2). Notably, both issues were later rediscovered independently by our AI-assisted analysis tools, demonstrating the value of LLMs in vulnerability discovery.

The remaining three findings emerged directly from our AI-driven testing methodologies. Our automated fuzzing framework identified a LevelDB construction issue that could cause panics under specific conditions (TOB-AVAX-3), while AI-powered static analysis revealed a timing vulnerability in the file permissions package (TOB-AVAX-4). The fifth issue was discovered when we adjusted the fuzzing framework to incorporate CPU and memory profiling, which identified the absence of request size limits in the API server (TOB-AVAX-5). This finding was then triaged using our AI-driven denial-of-service testing framework, which determined that HTTP request timeouts made this difficult to exploit in practice.

The AI-augmented approach proved particularly valuable for accelerating code comprehension and threat modeling across the massive codebase. AI tools successfully generated comprehensive attack trees, identified high-risk areas for focused manual review, and performed pattern matching against historical vulnerability datasets. As AI capabilities continue to evolve, we recommend that the Ava Labs team prepare for

increasingly sophisticated automated attacks by leveraging these same tools to enhance this codebase's defensive security practices.

## Recommendations

Based on the findings identified during this AI-focused security review, Trail of Bits recommends that Ava Labs take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed through direct fixes or broader refactoring efforts.
- **Establish AI-augmented security practices.** Integrate AI tools into regular security workflows and red team exercises to benefit from accelerated vulnerability discovery and threat modeling capabilities demonstrated in this engagement. See appendix B for more information.
- **Prioritize API endpoint hardening.** The successful AI-driven denial-of-service attacks against the RPC HTTP endpoint highlight the vulnerability of nodes that have this endpoint exposed. See appendix C for more information.

## Finding Severities and Categories

The following tables provide the number of findings by severity and category.

**EXPOSURE ANALYSIS**

| *Severity* | *Count* |
|---|---|
| High | 0 |
| Medium | 0 |
| Low | 0 |
| Informational | 5 |
| Undetermined | 0 |

**CATEGORY BREAKDOWN**

| *Category* | *Count* |
|---|---|
| Data Validation | 2 |
| Denial of Service | 2 |
| Timing | 1 |

# Project Goals

The engagement was scoped to provide a security assessment of the Ava Labs AvalancheGo codebase. Specifically, we sought to answer the following non-exhaustive list of questions:

- What denial-of-service vulnerabilities exist in AvalancheGo's API endpoints and network communication protocols?
- What consensus-breaking vulnerabilities exist in the snow primitives and Patricia tree implementations?
- How might the codec marshaling and unmarshaling implementations be exploited to cause stack overflow or resource exhaustion attacks?
- What data validation weaknesses exist in the MerkleDB synchronization mechanisms that could lead to processing inefficiencies?
- Are there race conditions or timing vulnerabilities in AvalancheGo's filesystem operations and database interactions?
- How could transaction validation and processing workflows in the AVM and PlatformVM be manipulated to bypass security controls?
- How might the fee system and gas mechanics be exploited to create economic attack vectors during high system loads?
- Could an attacker leverage AI tools to systematically identify and exploit critical vulnerabilities in the AvalancheGo codebase?
- What classes of denial-of-service attacks can AI agents autonomously execute against live AvalancheGo nodes?
- How effectively can AI-powered static analysis accelerate vulnerability discovery and threat modeling across AvalancheGo's more than 250,000 lines of code?
- What historical vulnerability patterns from the broader Go ecosystem are applicable to identifying security issues in AvalancheGo?
- How can AI-generated fuzz testing and automated exploit development enhance traditional security assessment methodologies for blockchain node implementations?

# Project Targets

The engagement involved reviewing and testing the following target.

### AvalancheGo

| | |
|---|---|
| Repository | https://github.com/ava-labs/avalanchego |
| Version | `9d1f232b756bff600b2abbec9e2cd592662e4a7d` |
| Type | Go |
| Platform | Web |

# Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Manual review.** Our traditional manual review process was enhanced through AI-powered code comprehension and a curated context-aware LLM containing critical components of the AvalancheGo source code and custom guidance, enabling more rapid understanding of the codebase.
    - **MerkleDB implementation and sync mechanisms.** We conducted a comprehensive analysis of MerkleDB data validation, synchronization logic between database instances, and potential for undefined behavior. Our review identified a minor denial-of-service vector in the `getRangeProof` method that can be triggered by requests with small response size limits (TOB-AVAX-2).
    - **Transaction validation and processing workflows.** We looked for transaction malleability concerns and examined cryptographic signature verification schemes and error handling across both the AVM and PlatformVM implementations. This included analysis of transaction processing between wallet clients and nodes and of block building and verification routines, as well as identification of possible consensus-breaking vulnerabilities in validator set management.
    - **Codec encoding/decoding security properties.** We analyzed the marshaling and unmarshaling functions for potential stack overflow vulnerabilities, focusing on the recursive nature of struct processing, and noted a lack of proper recursion limits in the `reflectcodec` implementation (TOB-AVAX-1).
    - **Consensus mechanisms and snow primitives.** We reviewed the Patricia tree implementation, snowball consensus parameters, and ProperVM logic with particular attention to clock skew implications and seed value generation for proposer selection.
    - **Fee system and gas mechanics.** We investigated transaction complexity calculations, gas system design, and fee adjustment mechanisms during high system loads to identify potential economic attack vectors.
- **AI-driven static analysis using historical vulnerability patterns.** We compiled a comprehensive primer document from 422 high- and medium-severity Go bug reports scraped from Solodit, creating a structured knowledge base of relevant, real-world security issues. This dataset was used to generate issue detection

prompts, security checklists for manual review, and high-level guidance to identify the highest-impact targets within the AvalancheGo codebase.

- **AI-driven fuzz test generation and validation pipeline.** We implemented an automated testing framework using the Claude Code SDK that continuously generates fuzz test cases, validates them with unit tests, and performs root cause analysis. The system identifies gaps in test coverage, generates targeted fuzz tests to identify insufficient data validation and arithmetic concerns, compiles and executes tests in iterative loops, and produces comprehensive reports including test status, profiling data, and security observations for manual triage.
- **AI agent denial-of-service testing against live nodes.** We deployed a testnet AvalancheGo node and created an autonomous AI agent specifically designed to execute denial-of-service attacks against both HTTP RPC and peer-to-peer (P2P) interfaces. The agent was equipped with custom diagnostic endpoints to monitor attack progress and achieved complete node unresponsiveness (17.36 s response times) through multi-endpoint bombardment and sustained pressure campaigns, demonstrating the practical threat posed by AI-enabled adversaries.
- **AI-generated Semgrep rules with intelligent result triage.** We used AI code generation tools to create AvalancheGo-specific Semgrep rules and deployed LLMs to assess the large volume of results by comparing findings to additional code context and generating proof-of-concept unit tests to validate potential vulnerabilities.
- **Custom Go compiler fork for arithmetic vulnerability detection.** We developed a novel approach for investigating integer overflow and underflow bugs by forking the Go compiler and adding additional checks at the intermediate-representation level. This enhanced compiler was integrated with our fuzzing harnesses in an effort to detect a broader class of arithmetic vulnerabilities, with filtering mechanisms to reduce false positives from expected overflow scenarios in built-in modules.

## Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Experimental nature and inherent limitations of AI-augmented security methodologies.** While our AI-powered analysis leveraged state-of-the-art tooling and novel approaches, these methodologies remain experimental and nondeterministic in nature. The AI-driven vulnerability detection, automated fuzz test generation, and pattern matching systems cannot guarantee comprehensive coverage of all potential security issues within their respective domains, and the

effectiveness of these tools may change drastically in the future, given the rapidly evolving capabilities of the underlying AI models.

- **Deprioritized manual review.** The AvalancheGo codebase exceeded our capacity for comprehensive manual analysis within the engagement timeframe, even with AI assistance. We strategically prioritized AI-powered automated testing methodologies over exhaustive manual review to maximize effective coverage across the system. As a result, large portions of the codebase received minimal manual security assessment, or none at all.

# Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

| ID | Title | Type | Severity |
|---|---|---|---|
| 1 | Unbounded recursion in codec allows stack overflow via deeply nested structs | Data Validation | **Informational** |
| 2 | Lack of lower bound on range proof request bytes limit enables denial of service | Denial of Service | **Informational** |
| 3 | LevelDB prone to panic from poor construction | Data Validation | **Informational** |
| 4 | TOCTOU in the perms package Create method | Timing | **Informational** |
| 5 | API server does not limit large request body sizes | Denial of Service | **Informational** |

# Detailed Findings

### 1. Unbounded recursion in codec allows stack overflow via deeply nested structs

| Severity: **Informational** | Difficulty: **High** |
|---|---|
| Type: Data Validation | Finding ID: TOB-AVAX-1 |
| Target: `codec/reflectcodec/type_codec.go` | |

**Description**

AvalancheGo uses a codec to marshal and unmarshal data. This codec is used in various components such as WARP, VMs, and the P2P system of Avalanche.

While the codec manager has a default maximum size of 256 KiB, it is in most cases instantiated with a maximum size of `math.MaxInt`, allowing large amounts of data to be marshaled/unmarshaled.

The issue lies in the lack of a recursion limit (see the example in figure 1.1). An attacker could craft a deeply nested struct that causes a stack overflow due to the recursive nature of the marshaling/unmarshaling functions in the codec.

```
case reflect.Struct:
	serializedFields, err := c.fielder.GetSerializedFields(value.Type())
	if err != nil {
		return err
	}
	for _, fieldIndex := range serializedFields { // Go through all fields of this
struct that are serialized
		if err := c.marshal(value.Field(fieldIndex), p, typeStack); err != nil
{ // Serialize the field and write to byte array
			return err
		}
	}
	return nil
```

*Figure 1.1: Example of recursive code responsible for the marshaling of a struct (codec/reflectcodec/type_codec.go#L414–L422)*

### Exploit Scenario

An attacker causes a denial-of-service by sending a deeply nested struct through the P2P network or other components using the codec. Any node attempting to unmarshal the malicious payload encounters a stack overflow and crashes. Since the codec is used throughout AvalancheGo, this affects multiple nodes processing the same data.

### Proof of Concept

The test shown in figure 1.2 constructs an approximately 45 MB nested struct that, during marshaling, will overflow the maximum stack limit of the Go runtime (set as 1 GB by default).

```
func TestCodecStackOverflow(t testing.TB, codec codecpkg.GeneralCodec) {
	type SliceNestedStruct struct {
		Children []SliceNestedStruct `serialize:"true"`
	}

	require := require.New(t)
	manager := codecpkg.NewManager(math.MaxInt)
	require.NoError(manager.RegisterCodec(0, codec))

	var buildPayload func(int) []SliceNestedStruct
	buildPayload = func(d int) []SliceNestedStruct {
		if d <= 0 {
			return []SliceNestedStruct{{}}
		}
		return []SliceNestedStruct{{
			Children: buildPayload(d - 1),
		}}
	}

	workingPayload := buildPayload(1000000)
	_, _ = manager.Marshal(0, workingPayload)
}
```

*Figure 1.2: Proof of concept showcasing the stack overflow due to deeply nested struct*

```
runtime: goroutine stack exceeds 1000000000-byte limit
runtime: sp=0x140201e0390 stack=[0x140201e0000, 0x140401e0000]
fatal error: stack overflow

runtime stack:
...
goroutine 22 gp=0x14000083500 m=6 mp=0x14000580008 [running]:
github.com/ava-labs/avalanchego/codec/reflectcodec.(*structFielder).GetSerializedFie
lds(0x140000b2d40, {0x104e21fc0, 0x104de34e0})

avalanchego/codec/reflectcodec/struct_fielder.go:52 +0x3e8 fp=0x140201e0390
sp=0x140201e0390 pc=0x104ceb098
github.com/ava-labs/avalanchego/codec/reflectcodec.(*genericCodec).marshal(0x1400009
2660, {0x104de34e0?, 0x14001ae6000?, 0x104ceacfc?}, 0x140000b2040, 0x0)
```

```
avalanchego/codec/reflectcodec/type_codec.go:414 +0x76c fp=0x140201e05e0
sp=0x140201e0390 pc=0x104cec7ec
github.com/ava-labs/avalanchego/codec/reflectcodec.(*genericCodec).marshal(0x1400009
2660, {0x104db23a0?, 0x14001ae6040?, 0x0?}, 0x140000b2040, 0x0)
...
```

*Figure 1.3: Stack overflow triggered after executing `TestCodecStackOverflow`*

### Recommendations

Short term, implement a proper recursion limit check for every field, for both marshaling and unmarshaling operations. The codec should track the current recursion depth and return an error when a maximum depth is exceeded to prevent stack overflow.

Long term, in code responsible for processing nested data, implement a maximum recursion limit where applicable to prevent resource exhaustion attacks. Additionally, carefully consider the `maxSize` value in the context of `NewManager` instantiation on a `GeneralCodec` instance: while it would not fix the core problem, a small `maxSize` would prevent such a stack overflow from happening.

### 2. Lack of lower bound on range proof request bytes limit enables denial of service

| Severity: **Informational** | Difficulty: **Low** |
| --- | --- |
| Type: Denial of Service | Finding ID: TOB-AVAX-2 |
| Target: `x/sync/network_server.go` | |

#### Description

Attackers can increase the computation required to respond to `SyncGetRangeProofRequest` requests by issuing a request with the maximum `KeyLimit` value and a small but nonzero `BytesLimit` value (e.g., 1).

This problem arises from an iterative reduction of the `KeyLimit` value and a lack of lower bound on the `BytesLimit` value in the `getRangeProof` method, shown in figure 2.1.

```
// TODO improve range proof generation so we don't need to iteratively
// reduce the key limit.
func getRangeProof(
	ctx context.Context,
	db DB,
	req *pb.SyncGetRangeProofRequest,
	marshalFunc func(*merkledb.RangeProof) ([]byte, error),
) ([]byte, error) {
	root, err := ids.ToID(req.RootHash)
	if err != nil {
		return nil, err
	}

	keyLimit := int(req.KeyLimit)

	for keyLimit > 0 {
		<SNIPPED>

		if len(proofBytes) < int(req.BytesLimit) {
			return proofBytes, nil
		}

		// The proof was too large. Try to shrink it.
		keyLimit = len(rangeProof.KeyChanges) / 2
	}
	return nil, ErrMinProofSizeIsTooLarge
}
```

*Figure 2.1: A subset of the `getRangeProof` method (`x/sync/network_server.go#L248–L291`)*

This method generates a proof for the required number of keys. If the resulting proof is bigger than the `BytesLimit` value, the method cuts the key limit in half and tries again until the division step returns zero, in which case it returns an `ErrMinProofSizeIsTooLarge` error. Given the lack of a lower bound on the `BytesLimit` value, attackers can force this iteration to run the maximum number of times.

### Exploit Scenario

An attacker saturates their rate limit with repeated range proof requests with the max `KeyLimit` value and a `BytesLimit` value of 1. As a result, the target server experiences twice as much load as intended, and response times are degraded.

### Recommendations

Short term, impose a lower bound on `BytesLimit`. For example, require that `BytesLimit` be greater than or equal to a constant multiplied by the `KeyLimit`. This constant would depend on the kind of data the Merkle database is storing, so it should be a configuration parameter rather than hard-coded.

Long term, revisit the `TODO` comment at the top of this method. If the iterative reduction of the `keyLimit` value can be removed, then this issue will be removed along with it.

### 3. LevelDB prone to panic from poor construction

| Severity: **Informational** | Difficulty: **High** |
| --- | --- |
| Type: Data Validation | Finding ID: TOB-AVAX-3 |
| Target: `database/leveldb/db.go#L329–L337` | |

**Description**

LevelDB's internal iterator object is prone to panics if it is improperly constructed. For developers creating their chains off of Avalanche, this may produce undefined behavior or affect availability.

For example, with the following test, the red-highlighted line of code will panic instead of allowing the user to use `Next()` and `Error()` messages to check for any errors, as expected:

```
func TestNewIteratorWithStartAndPrefix_Vulnerability(t *testing.T) {
	folder := t.TempDir()
	db, err := New(folder, nil, logging.NoLog{}, prometheus.NewRegistry())
	require.NoError(t, err)
	defer db.Close()

	// Add data and force compaction to create the conditions for the
vulnerability
	for i := 0; i < 1000; i++ {
		key := []byte(fmt.Sprintf("key%d", i))
		value := []byte(fmt.Sprintf("value%d", i))
		err := db.Put(key, value)
		require.NoError(t, err)
	}
	err = db.Compact(nil, nil)
	require.NoError(t, err)

	iter := db.NewIteratorWithStartAndPrefix([]byte("\xb0"), []byte("0"))
	defer iter.Release()
}
```

*Figure 3.1: Test demonstrating that problematic constructions of the `Iterator` are not sufficiently validated*

Reviewing the affected code, we can observe that the highlighted comparison would pass due to the first byte sequence being greater in sorted value than the second byte sequence, due to `{0xB0}` (start) being greater than `{0x30}` (prefix, from "`0`" value).

```
func (db *Database) NewIteratorWithStartAndPrefix(start, prefix []byte)
database.Iterator {
    iterRange := util.BytesPrefix(prefix)
    if bytes.Compare(start, prefix) == 1 {
        iterRange.Start = start
    }
    return &iter{
        db:       db,
        Iterator: db.DB.NewIterator(iterRange, nil),
    }
}
```

*Figure 3.2: Problematic constructions of the `Iterator` are not sufficiently validated. (database/leveldb/db.go#L329–L338)*

This would cause `NewIterator` to be called with a range of `[0xb0, 0x31)`. `NewIterator` will eventually reach a call to `tFiles.newIndexIterator` within `table.go`, which will trigger the panic.

Note that the call to `db.Compact` in figure 3.1 was necessary to trigger the vulnerability. Without this call, the construction will succeed without panicking, which may indicate that the issue is latently triggerable by a developer.

### Recommendations

Short term, add validation to LevelDB's `NewIteratorWithStartAndPrefix` method to ensure that a valid range is clamped to an empty range, or that an error is otherwise returned by `iterator.Error()` to allow developers to use the structure as intended. Alternatively, document the caveat in the function summary.

Long term, write fuzz tests for all outer-edge providers that can easily be tested. Ensure that all undefined behavior is fixed or documented for developer awareness.

### 4. TOCTOU in the perms package Create method

| Severity: **Informational** | Difficulty: **Medium** |
| --- | --- |
| Type: Timing | Finding ID: TOB-AVAX-4 |
| Target: `utils/perms/create.go#L11–L24` | |

#### Description

The `perms` package offers a method to create a file with a given set of permissions (the `Create` method, shown in figure 4.1). However, it is prone to time-of-check to time-of-use (TOCTOU) vulnerabilities because the `Stat`, `Chmod`, and `OpenFile` operations happen independently. This could allow an attacker to introduce undefined behavior into the application by changing the target file between execution of these commands.

```
func Create(filename string, perm os.FileMode) (*os.File, error) {
	if info, err := os.Stat(filename); err == nil {
		if info.Mode() != perm {
			// The file currently has the wrong permissions, so update them.
			if err := os.Chmod(filename, perm); err != nil {
				return nil, err
			}
		}
	} else if !errors.Is(err, os.ErrNotExist) {
		return nil, err
	}
	return os.OpenFile(filename, os.O_RDWR|os.O_CREATE|os.O_TRUNC, perm)
}
```

*Figure 4.1: The `Create` method, which is subject to TOCTOU vulnerabilities (`utils/perms/create.go#L12–L24`)*

This method also does not check whether the file to be created is a preexisting symbolic link, introducing a risk for its use as a primitive for privileged filesystem attacks across the system.

Note that this issue was marked as informational-severity, as this method is largely used to write CPU/memory profile data and an exploit is not expected to be impactful in its current state. However, future uses of this vulnerable method may increase the impact of this issue.

#### Exploit Scenario

The following scenarios are possible given the properties of the `Create` method:

- If the file already exists and its permissions are not the same as requested, the method will attempt to change the permissions and open the file.
    - An attacker can re-create the file prior to the call to `Chmod` in a symbolic link attack, causing the system to change permissions of another file on the system and truncate its contents.
- If the file does not exist, the system will attempt to create it with the given permissions.
    - An attacker can introduce the file with their own permissions between the calls to `Stat` and `OpenFile`, retaining the permissions they set for the new file (rather than those requested).

**Recommendations**

Short term, rewrite this function so that `OpenFile` is called first and `Stat` and `Chmod` are performed against the file handle returned by `OpenFile`. This will ensure that `Stat` and `Chmod` are not performed against a different file residing at the same path, but will operate on the originally opened file, avoiding TOCTOU concerns.

Additionally, add an optional check against symbolic links, which can be enforced for critical files.

Long term, ensure that filesystem code is considerate of using opened file handles rather than repetitive use of a file path. This will mitigate concerns of TOCTOU-related issues.

### 5. API server does not limit large request body sizes

| Severity: **Informational** | Difficulty: **Medium** |
| --- | --- |
| Type: Denial of Service | Finding ID: TOB-AVAX-5 |
| Target: `api/server/server.go#L67–79`, `api/server/server.go#L132–138` | |

#### Description

Although the AvalancheGo documentation indicates that nodes should not be publicly exposed without a front-facing gateway, in order to prevent denial-of-service attacks, the HTTP servers across the repository often enforce `ReadTimeout`, `ReadHeaderTimeout`, and `WriteTimeout` values.

The API server enforces these values but does not enforce a request body size. Subsequently, many API handlers perform an `io.ReadAll` call on request bodies and often perform JSON deserialization afterward with the `gorilla/rpc` JSON codec, which does not enforce size limits either.

As a result, the API server is noted to quickly exhaust memory in a concurrent large request body attack.

#### Recommendations

Short term, introduce a `MaxHeaderBytes` size to the HTTP server, and wrap request body reading in an `http.MaxBytesReader` with some reasonable upper limit.

Long term, ensure all HTTP servers enforce read and write timeouts to protect against Slowloris attacks, and enforce request header and body size limits to protect against large packet denial-of-service attacks. Where applicable, implement IP rate-limiting mechanisms.

# A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

| Vulnerability Categories | |
|---|---|
| **Category** | **Description** |
| **Access Controls** | Insufficient authorization or assessment of rights |
| **Auditing and Logging** | Insufficient auditing of actions or logging of problems |
| **Authentication** | Improper identification of users |
| **Configuration** | Misconfigured servers, devices, or software components |
| **Cryptography** | A breach of system confidentiality or integrity |
| **Data Exposure** | Exposure of sensitive information |
| **Data Validation** | Improper reliance on the structure or values of data |
| **Denial of Service** | A system failure with an availability impact |
| **Error Reporting** | Insecure or insufficient reporting of error conditions |
| **Patching** | Use of an outdated software package or library |
| **Session Management** | Improper identification of authenticated users |
| **Testing** | Insufficient test methodology or test coverage |
| **Timing** | Race conditions or other order-of-operations flaws |
| **Undefined Behavior** | Undefined behavior triggered within the system |

| Severity Levels | |
|---|---|
| **Severity** | **Description** |
| **Informational** | The issue does not pose an immediate risk but is relevant to security best practices. |
| **Undetermined** | The extent of the risk was not determined during this engagement. |
| **Low** | The risk is small or is not one the client has indicated is important. |
| **Medium** | User information is at risk; exploitation could pose reputational, legal, or moderate financial risks. |
| **High** | The flaw could affect numerous users and have serious reputational, legal, or financial implications. |

| Difficulty Levels | |
|---|---|
| **Difficulty** | **Description** |
| **Undetermined** | The difficulty of exploitation was not determined during this engagement. |
| **Low** | The flaw is well known; public tools for its exploitation exist or can be scripted. |
| **Medium** | An attacker must write an exploit or will need in-depth knowledge of the system. |
| **High** | An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue. |

# B. AI Usage

## Overview

This section describes how we used AI during this engagement to guide vulnerability discovery and testing, and it discusses other potential opportunities to apply AI/ML throughout the codebase.

## Approaches

While time constraints prevented us from developing a custom AI agent, we used Claude and Claude Code for manual analysis and the Claude Code SDK to perform automated analyses.

The following details our approaches taken at a high level. Additional considerations can be found in the Challenges subsection.

### 1. Onboarding through AI-powered analysis

#### Goals

- Accelerate auditor spin-up on the system under audit by performing forward-thinking analysis of system properties/invariants and design principles.
- Generate artifacts useful to subsequent audit steps.

#### Methodology

- Used Claude and Claude Code (manual usage). Claude was seeded with documentation and additional audit context to improve accuracy.
- Generated comprehensive big-picture explanations of system components, their interactions, and overall design patterns to accelerate auditor spin-up.
- Accelerated code comprehension and security analysis through targeted AI queries to quickly understand complex logic within the codebase.
    - Example: “What does the function `BuildBlock()` do, and what are its security implications?“
- Surfaced system properties and invariants to rapidly assess functionality and identify potential security implications across different components.
    - Example: “As a security engineer, what attack vectors should I consider when dealing with this API endpoint? What system properties or invariants are expected to be upheld?“

### Results

These efforts allowed us to rapidly investigate different portions of the codebase. This was especially valuable for code paths that were not directly under analysis but whose properties had implications for those that were.

Moreover, preemptive generation of system properties and invariants across code paths to be reviewed helped us gain an initial foothold in understanding a code path's design pattern implications.

## 2. Identifying high-value areas of concern

### Goals

- Validate suspected vulnerabilities.
- Discover additional vulnerabilities through a directed review of the codebase.

### Methodology

- Used Claude and Claude Code (manual usage). Claude was seeded with documentation and additional audit context to improve accuracy.
- Enumerated areas of the codebase with less testing coverage, which may warrant further review.
    - Example: "Review test suite coverage across the codebase. Enumerate a list of components that have not been tested against, or properties that were not evaluated within existing tests."
- Developed comprehensive threat models and attack trees using AI to systematically identify potential attack vectors across the AvalancheGo codebase.
    - These models mapped out potential exploitation paths and were used to prioritize vulnerabilities by likelihood and impact, enabling more focused audit efforts on the highest-risk components.
    - Example: "In the [`package_name`] package, identify security properties and invariants expected to be upheld. Consider cases of insecure usage of the package, misconfiguration, insufficient data validation, and undefined behavior that other components may not account for."
- Enhanced vulnerability detection through pattern matching by using AI to identify similar security issues across the codebase.
    - After discovering a vulnerability, we employed queries such as "I've found this bug. Are there more cases of it?" to systematically locate related instances and ensure comprehensive coverage of vulnerability classes.

#### Results

We were able to rapidly identify portions of code to review, quickly spin up on properties/invariants in a given code path, find areas less covered by tests, and perform rapid variant analysis of existing issues.

Coverage was significantly increased in various regions of the codebase. Classes of issues that were reviewed lightly in some paths were met with additional validation. Helper utilities were analyzed for many edge case bugs when considering the use of AvalancheGo as an SDK by other developers.

### 3. Denial-of-service attack vector evaluation

#### Goals

- Use AI agents to generate and autonomously conduct a variety of denial-of-service attack strategies against a local test node.

#### Methodology

- Deployed two AvalancheGo test nodes for controlled testing.
    - The first was a simple, unmodified Fuji testnet node that served as the stress testing target. The second was modified to disable outbound message throttling and include an MCP server, allowing the AI agent to control it during attacks on the target node's P2P interface.
- Created monitoring scripts and a custom web server to provide the attacking agent with health status, response times, and system logs, giving it additional context into the impact of each attack strategy.
- Leveraged agents to execute multiple denial-of-service strategies against the test node, recording observations and results.
    - Strategies include, but are not limited to, connection spam, flood attacks, payload attacks, malformed requests, multi-endpoint flooding, sustained attacks, and rate limit bypasses.

#### Results

The approaches and results of this effort are detailed in greater depth in appendix C.

This effort thoroughly covered a variety of denial-of-service attack vectors and was used to validate and assess the severity of TOB-AVAX-5.

### 4. Automated fuzzing

#### Goals

- Conduct automatic fuzz test generation for traditional security concerns, including improper arithmetic, insufficient data validation, and undefined behavior.

- Conduct automatic profiling of code paths for denial of service through fuzz test generation and CPU and memory profiling, with analysis performed over some finite period.

### Methodology

- Leveraged the Claude Code SDK to automate prompts.
- Automated the following high-level strategy:
    - Identify gaps in fuzz testing coverage (missing tests, unvalidated properties), automate writing the fuzz test and fixing identified gaps until compilation succeeds, run it for X seconds, and collect observations and insights from the test.
    - In the event of a failure, perform root cause analysis, trace upward to validate that the issue is reachable, and write a unit test to reproduce any findings.
    - Save all results into a report file.
    - Loop back to the start.
- Extended this strategy by encouraging Claude Code to consider classes of vulnerabilities stored in files in a "seeds" directory, initially seeded with classes of CWE vulnerabilities.
- Maximized coverage and mitigated hallucinations by analyzing outstanding fuzz testing ideas not previously documented in an "ideas" or "ideas-completed" folder, and updating them accordingly with target information, results, and observations as each test was executed.
- Applied temporary prompt tweaks to encourage production of fuzz tests that were not aimed at producing panics but would look for resource exhaustion attacks while performing CPU and memory profiling.

### Results

A `claude-fuzz` package was created for this automation. It was iteratively updated throughout the assessment, flagging various concerns in different iterations. After these updates, the package rediscovered issues we already discovered via manual review, such as TOB-AVAX-1, and identified new issues such as TOB-AVAX-3 and TOB-AVAX-5.

While the original prompts for some identified issues have since been overwritten throughout iterations during the engagement, the orchestrator script's source code has been provided as an artifact externally from this report.

### 5. Automated auditing using a judge advocate model

#### Goals

- Reduce false positive rates in AI-generated vulnerability reports through systematic verification.
- Implement a multi-stage review process that mirrors human security assessment workflows.
- Create a probabilistic scoring system for vulnerability claims to enable efficient triage.

#### Methodology

- Developed a three-stage adversarial review system using the Claude Code SDK to validate potential vulnerabilities through structured debate.
- Implemented a bug-hunting agent that generates structured vulnerability reports in JSON format with descriptions, reasoning, and code references.
- Implemented a judge advocate model that fact-checks code references and identifies logical flaws in vulnerability arguments.
- Implemented a reasoning judge that analyzes arguments and counterarguments to assign a probability score (0–1) for vulnerability likelihood.
- Established a configurable threshold system where only high-scoring vulnerabilities are escalated for human review.

#### Results

The judge advocate system reduced false positive rates while maintaining coverage of legitimate security concerns. The adversarial review process identified common AI hallucinations like misunderstanding code context and making unfounded assumptions about system behavior.

The probabilistic scoring enabled efficient triage by filtering low-confidence findings while highlighting issues that warranted manual investigation. This proved particularly valuable for distinguishing genuine potential security issues from superficial code patterns that can appear problematic without proper context.

### 6. Vulnerability pattern matching using historical bug datasets

#### Goals

- Leverage historical vulnerability data to identify similar security issues within the AvalancheGo codebase.

- Accelerate vulnerability discovery by applying known vulnerability classes from the broader Go ecosystem.
- Validate AI-driven vulnerability detection when provided with a comprehensive vulnerability context.

### Methodology

- Built upon the Solodit dataset established in approach 2 by extracting specific code patterns, function signatures, and structural characteristics from each vulnerability report.
- Analyzed each historical bug report to identify common vulnerability classes, attack vectors, and code patterns.
- Implemented systematic scanning where the agent searched AvalancheGo for code patterns that matched those found in the historical dataset.
- Conducted focused analysis on vulnerability classes relevant to distributed systems: denial-of-service vectors, data validation issues, concurrency problems, and resource exhaustion attacks.
- Generated security checklists and detection prompts based on the historical dataset to guide review processes.

### Results

The agent showed strong capability in recognizing similar code patterns when provided with historical context. However, it generated a significant number of false positives, requiring manual validation to distinguish genuine potential vulnerabilities from superficial pattern matches.

The approach worked best when combined with manual review, serving as a triage mechanism that directed auditors toward high-probability issue locations. The historical context improved the agent's understanding of subtle security implications that might not be apparent from code analysis alone.

## Challenges

In this section, we list challenges we encountered when using AI during the engagement. Under each main bullet point, we provide recommendations for responding to these challenges.

- Claude Code and other agents can be slow to respond to prompts.
    - Leverage the Claude Code SDK or whichever SDK is relevant to your agent to enable automation of prompts and workflows.
    - Note that parallelizing some requests in different sessions may be beneficial (e.g., writing unit tests in different Go packages), while other cases (e.g., updating an index) may not be due to race conditions and differing contexts.
- Agents may not consider all classes of vulnerabilities when directed to review some code.
    - Create different "seed files" containing different classes of issues to investigate or exclude issues based on previous coverage, and provide these files to the agent.
    - Continuously tweak investigative prompts to direct the agent to *explicitly* look for certain classes of issues it had not looked for previously, in directories it may not have previously searched.
    - Otherwise, prompt the agent to "check/test for other classes of vulnerabilities not yet evaluated" more broadly.
- Agents flag many false positives.
    - Include explicit language in prompts to encourage the agent to evaluate whether findings are false positives, such as "for any suspected finding, validate it is practically reachable," "write a unit test to validate the existence of the finding," and "perform root cause analysis and trace the issue upwards to ensure any higher-level checks do not invalidate the finding."
- Agents can lose context on large tasks or miss task items.
    - For suspicious code paths to investigate or suspected findings to validate, direct the agent to dump a verbose report to a file to be reingested later. This will ensure detailed context is saved after each workflow step prior to being potentially lost/deprioritized in the context window from subsequent workflow steps.

    - Ask the agent to dump out any coverage it achieved with the requests. This will allow the agent to track any tasks it does not complete, as hallucinations may result in incomplete coverage.

- Agents write bad tests or tests that simply mock the functionality to be tested.
    - Include language to prevent obvious cases of poorly written tests, tailored to the given language or test framework. For example, for Go, include a prompt like, "do not use any `recover()` statements in a way that mistakenly catches panics/test failures."
    - Include language encouraging the agent to write good tests, such as "ensure the test is well written and does not overly mock the code to be tested."
    - Include language encouraging the agent to verify the quality of its tests, such as "write a unit test to validate the findings of the fuzz test," before asking it to dump a report with the results.
- Agents are still hallucinating despite safeguards. They will surely continue to do so. Prompt tweaks and additional safeguards may mitigate some cases of hallucination, but they should not be expected to completely prevent it. Nonetheless, the following actions could help reduce and mitigate hallucination.

  Anticipate hallucinations resulting in erroneous file overwrite by isolating output files. Indicating in your prompt that the agent should "append results to [file path]" may result in an overwrite rather than an append, resulting in the loss of your progress. Isolating your output files instead will prevent this type of hallucination.
    - Use absolute file paths (if using the SDK) to avoid hallucinations caused by relative file path usage after the agent changes the current working directory.
    - Use a throwaway environment or sandbox for testing. Do not use a critical machine with sensitive data on it.

## Results Summary

The AI-augmented security assessment demonstrates AI's value as a force multiplier for code comprehension, threat modeling, and vulnerability pattern recognition, significantly accelerating auditor onboarding.

The judge advocate model reduced false positives through adversarial validation, while historical vulnerability datasets enhanced pattern recognition. However, both approaches generated substantial false positives requiring manual validation, confirming AI's role as a triage tool rather than a definitive vulnerability detection mechanism.

AI-driven denial-of-service testing achieved the most tangible results, with autonomous agents successfully causing complete node unresponsiveness, while producing the smallest

number of false positives. This demonstrates AI's immediate practical threat as an adversarial tool while highlighting critical defensive gaps in API endpoint protection.

AI-driven automated fuzzing and vulnerability detection showed promise but suffered from hallucination issues and context limitations. These limitations required additional consideration for techniques such as prompt engineering, safeguards, context window seeding, and additional validation passes. Despite such efforts, AI-driven analyses are expected to continue to produce a portion of uninteresting results. We found this acceptable in these cases, as we periodically identified some interesting results from the output.

All approaches were most effective when combined with manual review, serving to prioritize high-risk areas and accelerate traditional security workflows rather than replace human expertise.

## Future Considerations

The following recommendations provide concrete and speculative design patterns for Ava Labs to integrate AI into its workflows to achieve greater testing depth and enhance general system capabilities:

- **Custom agent development.** Future engagements should prioritize developing custom AI agents tailored to specific codebases and vulnerability classes, moving beyond general-purpose tools to specialized security assessment frameworks with domain-specific knowledge and validation capabilities. The use of custom agents can enable the team to do the following:
    - Write planners, code scanning agents, and other agents with more concrete guardrails to ensure the task does not deviate from the expected one.
    - Rely on concrete logic rather than an LLM by implementing more specialized tools/tasks. For instance, integrating LLM tools that operate against a live Avalanche node will give it more specialized capabilities and minimize hallucinations.
    - Integrate the team's own knowledge graphs with weights, or leverage retrieval-augmented generation (RAG) to connect vector databases with custom datasets to better guide the agent or contextualize its tasks.
- **Dynamic testing integration.** Expanding the use of AI beyond static analysis to incorporate AI-driven dynamic testing harnesses, intelligent test case generation, and automated exploit development could significantly enhance vulnerability discovery capabilities.
    - Integrate dynamic testing as a periodic scan or task to be run occasionally (e.g., as part of internal audit cycles).

    - Be mindful of its hallucinating results and attempt to use a judge model where possible. Manually review all results.
    - Be mindful of billing charges accrued due to public model use.
- **Enhanced prompting techniques.** Expand scanning and fuzzing prompt techniques to improve AI accuracy and coverage, developing more sophisticated prompt engineering approaches that reduce false positives and increase vulnerability detection rates.
- **Parallelization and scaling.** Implement centralized task distribution systems where ideas for fuzz testing can be centralized, but fuzz test writing and execution can be parallelized for speed. Set appropriate usage limits when using Claude Code and similar services, as costs can escalate quickly during large-scale analysis.
- **Data privacy and security.** Do not feed any private code or data to AI services without thoroughly reading their terms of service. Ensure that PII and sensitive organizational data are not incorporated into AI training models.
- **Defensive AI applications.** Explore AI applications for defensive purposes, including automated backdoor detection, sensitive information leak identification, dependency vulnerability scanning, and real-time attack pattern recognition in security monitoring systems.
- **Hybrid human–AI workflows.** Develop more sophisticated integration between AI analysis and human expertise, including AI-generated security checklists, automated report generation, and intelligent prioritization systems that enhance rather than replace traditional security assessment methodologies. These approaches can be leveraged to accomplish the following:
    - Enhance internal threat modeling and rapid risk assessments.
    - Assist in raising invariants and properties of the system, as well as potential concerns during peer review or internal audit cycles.
    - Identify and iterate upon gaps in incident response plans, operational policies and procedures, and awareness and training materials.

# C. AI Agent Denial of Service Attempts

To evaluate AvalancheGo's resilience against denial-of-service attacks, we deployed a local AvalancheGo node and configured it for automated stress testing. Our AI agent successfully executed autonomous denial-of-service attacks against this node, achieving complete unresponsiveness in multiple scenarios. The agent developed six distinct attack vectors and identified two highly effective approaches: multi-endpoint bombardment (10.06 s response times with empty responses) and sustained pressure campaigns (17.36 s response times with complete unresponsiveness). These results demonstrate the practical threat posed by AI-enabled adversaries and validate the critical importance of proper network access controls for AvalancheGo nodes.

The following outlines a manually curated subset of the agent's automatically generated documentation.

- Connection spam attack
    - Strategy: Overwhelm the node with rapid connection cycling to exhaust connection pools or file descriptors by creating 200 rapid HTTP connections to the health endpoint with 0.01 s delays between requests.
    - Results: Health response time was 0.84 s, and API response time was 0.38 s with status remaining true. This attack was completely ineffective, showing no significant impact on node performance, suggesting robust connection pooling and handling mechanisms.
- Flood attack
    - Strategy: Overwhelm request handling capacity by launching 500 concurrent `info.getNodeID` requests plus 50 health checks simultaneously to exhaust the node's ability to process requests.
    - Results: Health response time degraded significantly to 8.6 s (4.8 times slower than the baseline) while API response time remained normal at 0.38 s, with status still true. This attack was partially effective at degrading health endpoint performance, but the node remained stable overall.
- Payload attack
    - Strategy: Consume memory and processing resources with oversized requests by sending 20 requests containing 1 MB strings to the `admin.setLogLevel` endpoint.
    - Results: Health response time returned to normal at 0.96 s, and API response time remained at 0.38 s with the status true. This attack was ineffective, and

the node quickly recovered due to the HTTP request timeout's cutting off oversized payloads before they could be completely transferred.

- Malformed attack
    - Strategy: Cause parsing errors and crashes with malformed requests by sending 400 malformed requests, including broken JSON, 10 K–character method names, and deeply nested objects.
    - Results: Health response time increased moderately to 2.95 s (1.6 times slower than the baseline) and API response time increased slightly to 0.46 s, with status remaining true. This attack was partially effective, causing moderate performance degradation due to JSON parsing overhead.
- Multi-endpoint attack
    - Strategy: Simultaneously bombard all API endpoints to overwhelm routing and processing by hitting six different endpoints (info, health, admin, P, X, C/rpc) with 600 total requests distributed across all endpoints.
    - Results: Health response time reached 10.06 s (5.6 times slower than the baseline, exceeding the 10 s threshold) and API response time reached 5.00 s (8.9 times slower than the baseline), with HTTP status 000 and empty responses returned. This attack was highly effective, achieving both victory conditions of response times over 10 seconds and unhealthy status.
- Sustained attack
    - Strategy: Maintain continuous high-intensity pressure across multiple attack vectors by executing 10 waves of combined attacks, with each wave containing 200 flood requests, 250 multi-endpoint requests, and 100 connection spam attempts for a total of 3,500 requests.
    - Results: Health response time reached 17.36 s (9.6 times slower than the baseline, far exceeding the 10 s threshold), and API response time reached 5.00 s (8.9 times slower than the baseline), with HTTP status 000 and complete unresponsiveness indicated by empty responses. This attack achieved maximum effectiveness, causing complete unresponsiveness that far exceeded victory conditions.

These results revealed that spamming multiple API endpoints simultaneously was the most effective method. Sustained attacks prevented the node from recovering, while short bursts caused only minor performance degradation. Connection spam and large payload attacks proved ineffective in practice, while malformed requests caused only moderate processing overhead without major impact.

These results confirm that the warnings printed during the CLI installation script (reproduced below) are critical; it is trivial for a nontechnical attacker to use AI agents to conduct denial-of-service attacks against nodes with the RPC ports exposed.

> *If a firewall or other form of access control is not provided, your node will be open to denial-of-service attacks. The Node API server is not designed to defend against them! Make sure you configure the firewall to only let through RPC requests from known IP addresses!*

P2P layer testing required a modified attacking node to satisfy handshake requirements and enable programmatic control by the AI agent. The agent executed various P2P attack strategies including connection establishment, large message flooding, and sustained pressure campaigns, revealing the network layer's resilience under automated adversarial conditions. A curated subset of its results is provided below:

- Connection establishment
    - Strategy: Create multiple test peer connections from the attacking node to establish a foundation for message flooding attacks.
    - Results: Successfully created six test peer connections to the target node. Connection attempts beyond this limit were throttled by the target node's connection-limiting mechanisms, demonstrating that connection-level protections were functioning as designed.
- Large message flooding
    - Strategy: Overwhelm the target node with high-volume message flooding using 65 to 256 KB payloads across established peer connections.
    - Results: Successfully transmitted more than 31,500 messages, totaling approximately 7 GB of P2P traffic over 30 minutes. Attack operations exhibited 5 to 30 minute completion times compared to normal sub-second response times, suggesting processing strain on the target node and flagging this result for manual analysis.
- Sustained pressure campaign
    - Strategy: Maintain continuous high-intensity message flooding across multiple concurrent operations to prevent target node recovery.
    - Results: Executed 20 concurrent flood operations, sending 200 messages per operation per test peer, achieving a sustained attack duration of more than 30 minutes with consistent message delivery rates exceeding 95%. The target node demonstrated graceful degradation under load without crashing.

- Memory pressure testing
    - Strategy: Consume target node memory resources by flooding the node with maximum-size messages to identify resource exhaustion vulnerabilities.
    - Results: Successfully delivered more than 24,000 messages at 256 KB each, representing approximately 6.1 GB of payload data processed by the target node. Processing delays increased significantly, but no crashes or memory exhaustion conditions were observed.

The sustained message flooding attacks initially suggested a vulnerability when the agent measured 4 MB/s throughput versus the expected 3.07 MB/s limit. This agent generated a technical write-up suggesting missing time-based rate limits. However, manual review revealed that robust bandwidth throttling was already implemented with token bucket algorithms, and the observed 30% throughput variance fell within expected ranges for burst allowances.

This campaign demonstrated that AI tools excel at generating security leads and accelerating analysis, but can misinterpret complex defensive mechanisms like token bucket algorithms. The agent's ability to flag anomalous patterns and generate technical hypotheses proved valuable for directing expert attention. However, generating accurate final conclusions required manual guidance and review.

# About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up to date with our latest news and announcements, please follow @trailofbits on X or LinkedIn, and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

**Trail of Bits, Inc.**
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

# Notices and Remarks

## Copyright and Distribution



## Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.