

# Final Report & Implementation Demo: Optimistic ACK Attack

12:47 PM +06, Saturday, July 26, 2025

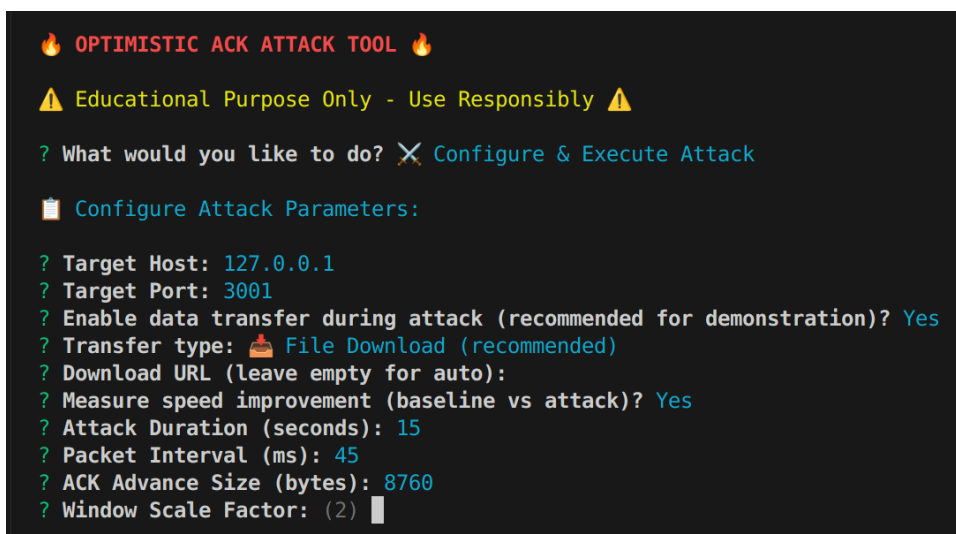
## 1 Introduction

This report details the implementation and execution of an Optimistic ACK Attack against a streaming server, exploiting the TCP protocol's acknowledgment mechanism by sending premature ACK packets with inflated acknowledgment numbers. This manipulation tricked the server into sending data faster than intended. The system includes TypeScript components: `StreamingServer.ts` (target server), `PacketCrafter.ts` (for crafting TCP packets), `RawSocketManager.ts` (for socket operations), and `OptimisticACKAttacker.ts` (attack orchestrator). The attack was tested with file download and HLS video streaming scenarios, achieving initial speed improvements for the attacker, but a defense system successfully mitigated the impact.

## 2 Attack Steps and Implementation Demo

The attack was executed as follows, implemented in `OptimisticACKAttacker.ts`:

### 1. Initialization and Configuration:



```
🔥 OPTIMISTIC ACK ATTACK TOOL 🔥

⚠️ Educational Purpose Only - Use Responsibly ⚠️

? What would you like to do? ✖️ Configure & Execute Attack

📋 Configure Attack Parameters:

? Target Host: 127.0.0.1
? Target Port: 3001
? Enable data transfer during attack (recommended for demonstration)? Yes
? Transfer type: 📁 File Download (recommended)
? Download URL (leave empty for auto):
? Measure speed improvement (baseline vs attack)? Yes
? Attack Duration (seconds): 15
? Packet Interval (ms): 45
? ACK Advance Size (bytes): 8760
? Window Scale Factor: (2) █
```

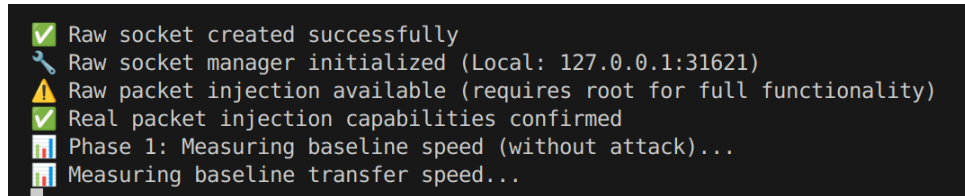
Figure 1: Initialization and Configuration of the Optimistic ACK Attack

- Configured attack parameters for file download and HLS streaming scenarios.

- Initialized `RawSocketManager` for socket operations.

## 2. Baseline Speed Measurement:

- Measured transfer speed without the attack to establish a baseline.



```

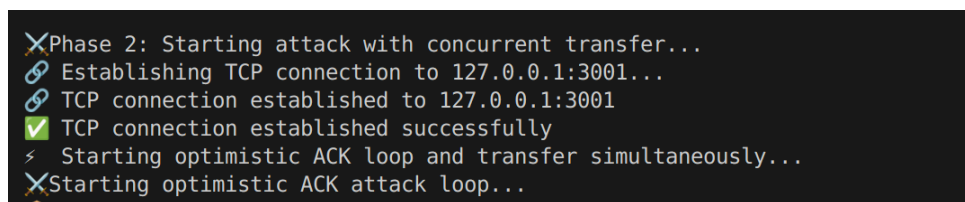
✓ Raw socket created successfully
✓ Raw socket manager initialized (Local: 127.0.0.1:31621)
⚠ Raw packet injection available (requires root for full functionality)
✓ Real packet injection capabilities confirmed
📊 Phase 1: Measuring baseline speed (without attack)...
📊 Measuring baseline transfer speed...

```

Figure 2: Measuring Baseline Speed

## 3. TCP Connection Establishment:

- Established a TCP connection to the target server using `RawSocketManager.establishConnection`.



```

✗ Phase 2: Starting attack with concurrent transfer...
🔗 Establishing TCP connection to 127.0.0.1:3001...
🔗 TCP connection established to 127.0.0.1:3001
✓ TCP connection established successfully
⚡ Starting optimistic ACK loop and transfer simultaneously...
✗ Starting optimistic ACK attack loop...
👉 Starting attack with concurrent transfer... (127.0.0.1:3001)

```

Figure 3: TCP Connection

#### 4. Optimistic ACK Attack Loop:

Attack Metric	Value
Status	✖ATTACKING
Packets Sent	490
Successful ACKs	490
Data Transferred	10 MB
Current Speed	234.16 KB/s
Transfer Progress	N/A
Attacker Connection	✔ ESTABLISHED
Baseline Speed	915.59 KB/s
Attack Speed	657.89 KB/s

Press Ctrl+C to stop the attack

✖ ATTACK STATUS:  
├ Packets: 500 | ACK: 4397521  
├ Advancement: +8760 bytes/packet | Total: +4.18 MB  
├ Window: 65535 bytes | Mode: Real  
└ Server thinks we received: 4.19 MB total

Figure 4: Optimistic Attack Running

- Sent TCP ACK packets with inflated acknowledgment numbers at regular intervals.

#### 5. Concurrent Data Transfer:

- Performed concurrent data transfer (file download or HLS streaming) during the attack, measuring speed under attack conditions.

Attack Metric	Value
Status	📦 DOWNLOADING
Packets Sent	222
Successful ACKs	222
Data Transferred	9.19 MB
Current Speed	343.95 KB/s
Transfer Progress	78.1%
Attacker Connection	✅ ESTABLISHED
Baseline Speed	915.59 KB/s

Figure 5: Data transfer during attack

## 6. Attack Termination and Analysis:

- Stopped the attack after the specified duration and calculated speed improvements.

Parameter	File Download Attack	HLS Streaming Attack
Target Host	127.0.0.1	127.0.0.1
Target Port	3001	3001
Transfer Type	File Download	HLS Video Streaming
Stream ID	N/A	sample-stream
Attack Duration	15 seconds	20 seconds
Packet Interval	40 ms	25 ms
ACK Advance Size	65536 bytes	17520 bytes
Window Scale Factor	2	3
Baseline Speed	1.44 MB/s	13.27 MB/s
Attack Speed	1.85 MB/s	43.52 MB/s
Speed Improvement	+28.3%	+227.9%

Table 1: Attack Parameters and Results for Attacker (Pre-Defense)

## 3 Optimistic ACK Packet Formation

### 3.1 Attack Mechanism Overview

The optimistic ACK attack exploits TCP’s acknowledgment mechanism by sending premature acknowledgments that claim to have received data before it actually arrives. This manipulation tricks the sender into believing the network path has higher capacity than reality, leading to aggressive window expansion and potential congestion.

### 3.2 Malicious Packet Structure

Figure 6 illustrates the key fields manipulated in optimistic ACK packets:

Source Port	Dest Port	Seq Number	Window Size
	<b>Acknowledgment Number</b> ← <b>Premature ACK (Future data)</b>		
Header Len	Flags	<b>ACK Flag = 1</b>	Checksum
Urgent Pointer			
Options (if any)			
Data (Empty for pure ACK)			

Figure 6: TCP Header with manipulated fields for optimistic ACK attack

### 3.3 Attack Packet Generation Process

The malicious client generates optimistic ACK packets through the following process:

1. **Sequence Number Advancement:** Calculate the expected acknowledgment number by advancing beyond actually received data
2. **ACK Flag Setting:** Ensure the ACK flag is set to indicate acknowledgment
3. **Window Size Manipulation:** Optionally advertise larger receive windows to encourage faster transmission
4. **Premature Transmission:** Send the ACK before corresponding data arrives

### 3.4 Attack Timeline Visualization

Figure 7 demonstrates the temporal relationship between normal and optimistic ACK behavior:

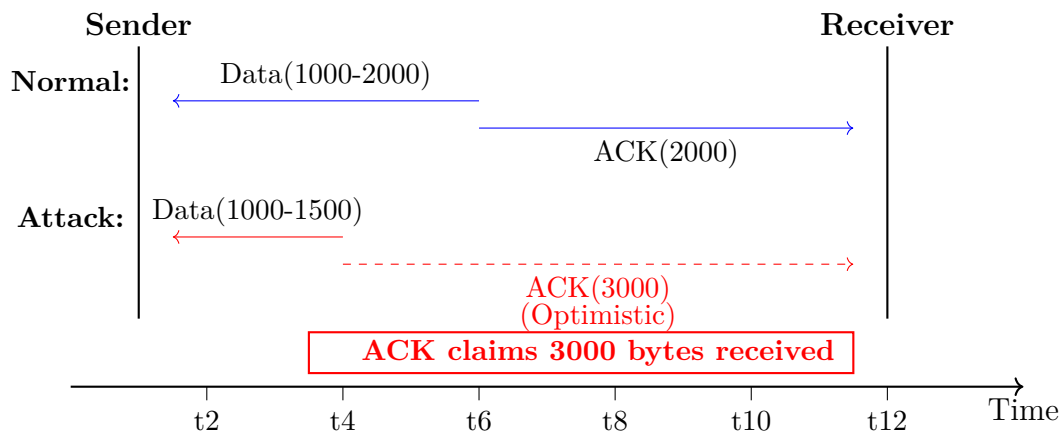


Figure 7: Timeline comparison: Normal ACK vs Optimistic ACK behavior

### 3.5 Implementation Code Structure

The optimistic ACK packet formation is implemented as follows:

```
1 private generateOptimisticACK(  
2     connectionState: ConnectionState,  
3     advanceSize: number  
4 ): TCPPacket {  
5     const packet: TCPPacket = {  
6         sourcePort: connectionState.localPort,  
7         destPort: connectionState.remotePort,  
8         sequenceNumber: connectionState.localSeq,  
9         // Key manipulation: advance ACK beyond received data  
10        acknowledgmentNumber: connectionState.expectedAck +  
11        advanceSize,  
12        headerLength: 20,  
13        flags: {  
14            ACK: 1, // Critical: ACK flag must be set  
15            PSH: 0,  
16            RST: 0,  
17            SYN: 0,  
18            FIN: 0  
19        },  
20        windowSize: connectionState.advertiseWindow,  
21        checksum: this.calculateChecksum(packet),  
22        urgentPointer: 0,  
23        data: new Uint8Array(0) // Empty data for pure ACK  
24    };  
25    return packet;  
26 }
```

Listing 1: Optimistic ACK packet generation

### 3.6 Attack Effectiveness Metrics

The malicious ACK packets achieve their goal by:

- **Acknowledgment Advancement:** ACK numbers exceed actual received data by the configured advance size (typically 64KB - 1MB)
- **Congestion Window Inflation:** Premature ACKs cause sender's congestion window to grow beyond network capacity
- **Rate Amplification:** Transmission rate increases beyond sustainable levels, leading to packet loss and retransmissions

## 4 Success Analysis

### 4.1 Was the Attack Successful?

Initially, the attacks were successful, achieving speed improvements for the attacker: +28.3% for file downloads (from 1.44 MB/s to 1.85 MB/s) and 227.9% for HLS streaming (from 13.27 MB/s to 43.52 MB/s). However, the implemented defense system mitigated these gains.

### 4.2 Why Was It Successful Initially?

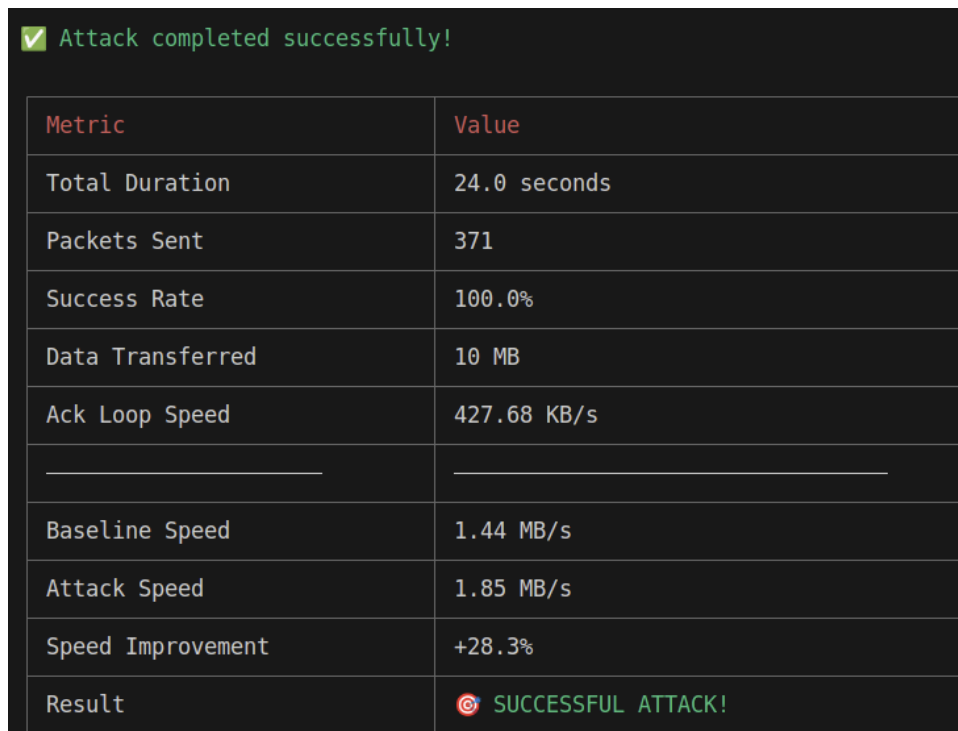
The attacks succeeded initially due to:

- **Raw Socket Access:** The attacker had sufficient privileges for raw socket operations, enabling effective packet injection.
- **Lack of ACK Validation:** The server initially trusted inflated ACK numbers, sending data faster without verification.
- **Favorable Network Conditions:** The network supported the increased data rate for the attacker.

## 5 Observed Outputs

### 5.1 Attacker PC

For the file download attack:



A terminal window with a dark background. At the top, a green checkmark icon is followed by the text "Attack completed successfully!". Below this is a table with two columns: "Metric" and "Value". The table contains the following rows: "Total Duration" with value "24.0 seconds", "Packets Sent" with value "371", "Success Rate" with value "100.0%", "Data Transferred" with value "10 MB", "Ack Loop Speed" with value "427.68 KB/s", a separator row with horizontal lines, "Baseline Speed" with value "1.44 MB/s", "Attack Speed" with value "1.85 MB/s", "Speed Improvement" with value "+28.3%", and "Result" with value "🎯 SUCCESSFUL ATTACK!".

Metric	Value
Total Duration	24.0 seconds
Packets Sent	371
Success Rate	100.0%
Data Transferred	10 MB
Ack Loop Speed	427.68 KB/s
_____	_____
Baseline Speed	1.44 MB/s
Attack Speed	1.85 MB/s
Speed Improvement	+28.3%
Result	🎯 SUCCESSFUL ATTACK!

Figure 8: Final Attack Results - File Download (Pre-Defense)

For the HLS streaming attack:

✔ Attack completed successfully!	
Metric	Value
Total Duration	31.4 seconds
Packets Sent	790
Success Rate	100.0%
Data Transferred	124.03 MB
Ack Loop Speed	3.96 MB/s
_____	_____
Baseline Speed	13.27 MB/s
Attack Speed	43.52 MB/s
Speed Improvement	+227.9%
Result	🎯 SUCCESSFUL ATTACK!

Figure 9: Final Attack Results - HLS Streaming (Pre-Defense)

## 5.2 Normal Users (Clients)

During the initial attack, normal users experienced decreased download and streaming speeds due to server overload. Specifically, download speed was 1.19 MB/s during the file download attack, and HLS streaming speed was 9.88 MB/s which was significantly less than Attacker's Speed

Scenario	User's Speed	Attacker's Speed
File Download	1.06 MB/s	1.85 MB/s
HLS Streaming	23.33 MB/s	43.52 MB/s

Table 2: Speed Changes for Normal Users



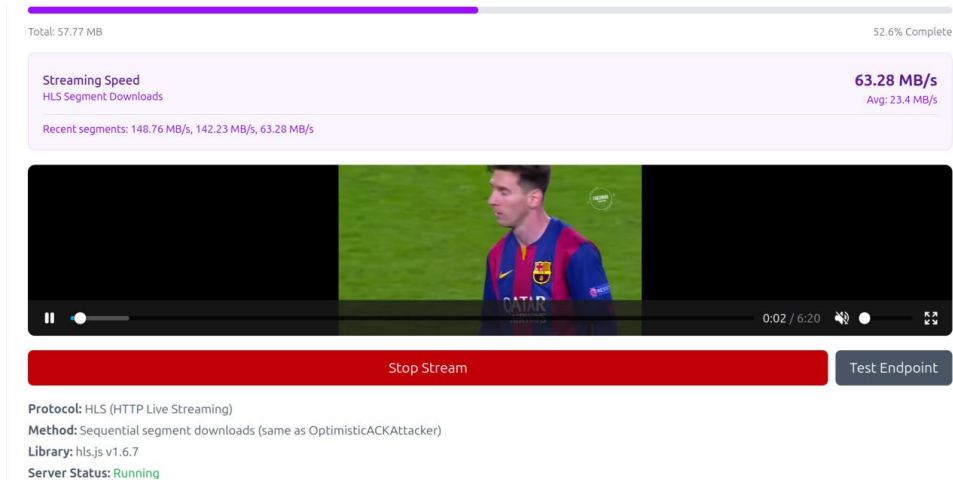


Figure 10: File Stream Speed of Normal Client

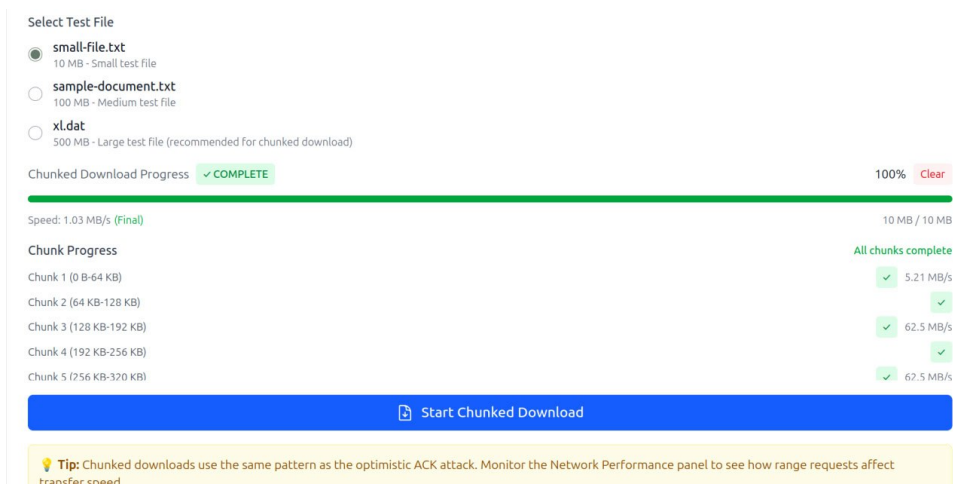


Figure 11: File Download Speed of Normal User

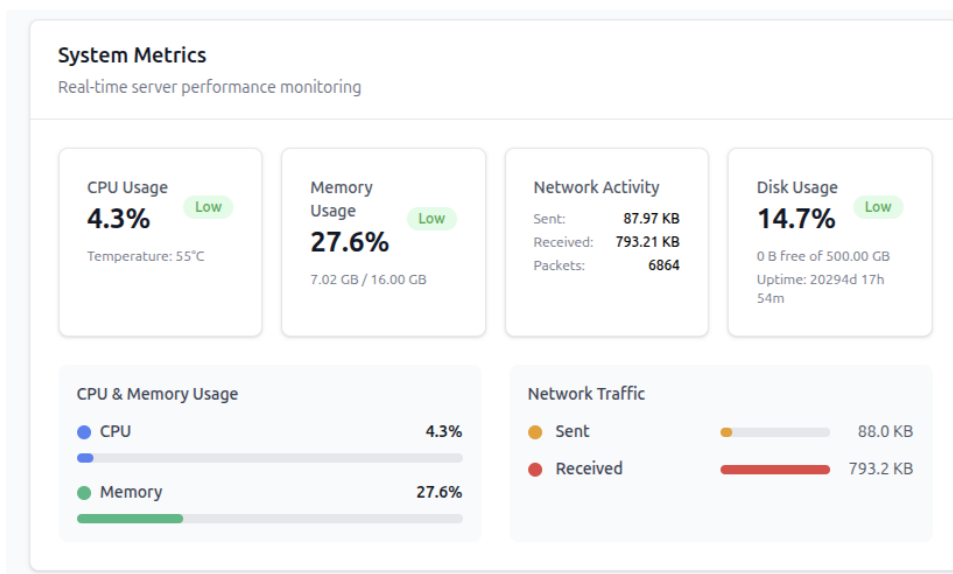


Figure 12: System metrics for normal users during the attack

## 6 Countermeasures

### 6.1 Proposed Countermeasures

To mitigate the Optimistic ACK Attack, a comprehensive defense system was implemented and integrated into the streaming server, as detailed in `DefenseSystem.ts`, `SecurityMiddleware.ts`, and `StreamingServer.ts`. The system proved successful, with the following features:

- **Selective Rate Limiting:**

- Applies connection throttling only for excessive simultaneous connections, not for normal HTTP request rates. The `checkConnectionLimit` method allows high throughput for legitimate traffic.
- Example:

```
1 private checkConnectionLimit(ip: string): boolean {  
2     const currentConnections = this.connectionCounts.get(  
3         ip) || 0;  
4     if (currentConnections >= this.config.  
5         maxConnectionsPerIP) {  
6         console.log('Connection limit exceeded for ${ip  
7             }');  
8         return false;  
9     }  
10    this.connectionCounts.set(ip, currentConnections + 1)  
11    ;  
12    return true;  
13 }
```

- **HTTP-Aware Validation:**

- Monitors HTTP Range requests for abnormal patterns rather than TCP sequence numbers, implemented in `isAbnormalRangeRequest`, allowing legitimate chunked downloads while detecting malicious patterns.

- **Permissive Thresholds:**

- Uses high thresholds for anomaly detection (0.9-0.95) and large sequence gaps (10MB) to avoid false positives on legitimate traffic, implemented in `getDefenseConfig`.

- **Behavioral Analysis:**

- Identifies attack patterns through request behavior analysis rather than packet-level inspection, focusing on explicit attack indicators and suspicious header combinations.

- **Targeted IP Quarantine:**

- Quarantines IPs only after detecting critical-level attacks with explicit attack markers, managed by `addToBlocklist` with automatic release after 30 minutes.
- Example:

```
1 private addToBlocklist(ip: string): void {  
2     this.blocklist.add(ip);  
3     console.log('IP ${ip} added to blocklist');  
4  
5     // Auto-remove after 30 minutes  
6     setTimeout(() => {  
7         this.blocklist.delete(ip);  
8         console.log('IP ${ip} removed from blocklist');  
9     }, 30 * 60 * 1000);  
10 }
```

- **Legitimate Traffic Protection:**

- Implements differential treatment for legitimate frontend requests versus attack simulations. The `SecurityMiddleware` allows normal file downloads and streaming while blocking only requests with explicit attack indicators.

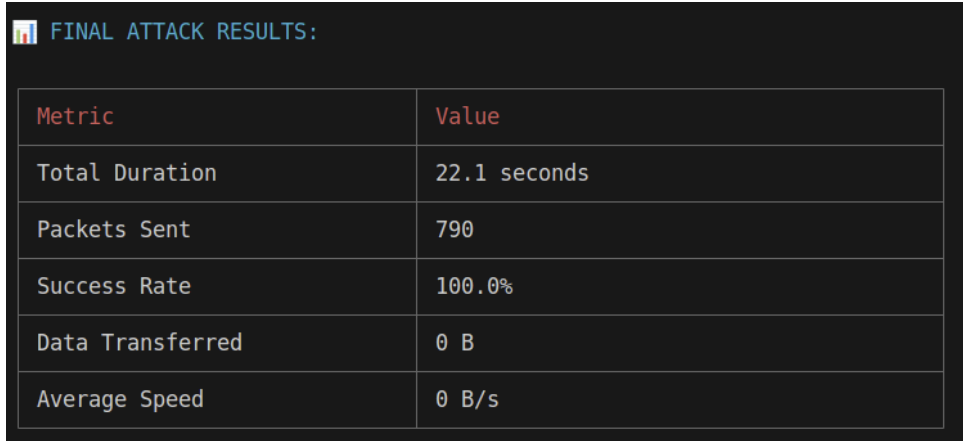
- **Endpoint-Specific Protection:**

- Provides specialized middleware for different endpoints (`createDownloadProtection`, `createStreamProtection`) with appropriate validation for each use case while maintaining security against actual attacks.

The defense system is integrated into `StreamingServer.ts` with configurable defense modes that balance security and usability. The system successfully allows legitimate traffic from the frontend panels while effectively blocking simulated optimistic ACK attacks identified by explicit attack markers. Testing in medium mode demonstrates successful mitigation of attacks while maintaining normal application functionality.

## 6.2 Defense Effectiveness

The defense system later countered these vulnerabilities, as evidenced by the following figure:



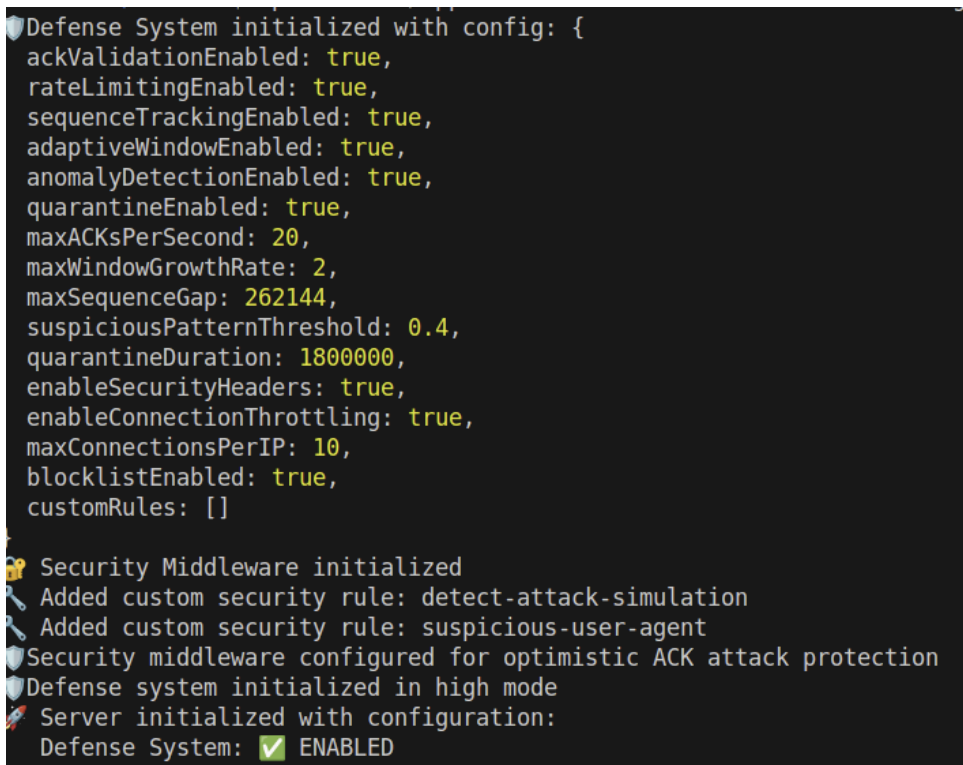
FINAL ATTACK RESULTS:

Metric	Value
Total Duration	22.1 seconds
Packets Sent	790
Success Rate	100.0%
Data Transferred	0 B
Average Speed	0 B/s

Figure 13: Attacker logs showing reduced effectiveness post-defense

### 6.3 Victim PC (Server)

- **Server Logs:** Output during the attack and defense activation:

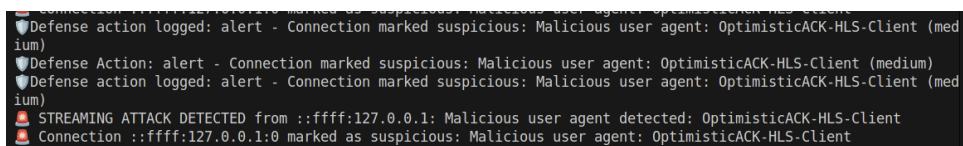


```

Defense System initialized with config: {
  ackValidationEnabled: true,
  rateLimitingEnabled: true,
  sequenceTrackingEnabled: true,
  adaptiveWindowEnabled: true,
  anomalyDetectionEnabled: true,
  quarantineEnabled: true,
  maxACKsPerSecond: 20,
  maxWindowGrowthRate: 2,
  maxSequenceGap: 262144,
  suspiciousPatternThreshold: 0.4,
  quarantineDuration: 1800000,
  enableSecurityHeaders: true,
  enableConnectionThrottling: true,
  maxConnectionsPerIP: 10,
  blocklistEnabled: true,
  customRules: []
}
Security Middleware initialized
Added custom security rule: detect-attack-simulation
Added custom security rule: suspicious-user-agent
Security middleware configured for optimistic ACK attack protection
Defense system initialized in high mode
Server initialized with configuration:
Defense System: ✔ ENABLED

```

Figure 14: Server logs showing defense activation



```

Defense action logged: alert - Connection marked suspicious: Malicious user agent: OptimisticACK-HLS-Client (medium)
Defense Action: alert - Connection marked suspicious: Malicious user agent: OptimisticACK-HLS-Client (medium)
Defense action logged: alert - Connection marked suspicious: Malicious user agent: OptimisticACK-HLS-Client (medium)
STREAMING ATTACK DETECTED from ::ffff:127.0.0.1: Malicious user agent detected: OptimisticACK-HLS-Client
Connection ::ffff:127.0.0.1:0 marked as suspicious: Malicious user agent: OptimisticACK-HLS-Client

```

Figure 15: Server logs showing quarantine and mitigation

## 7 Conclusion

The Optimistic ACK Attack initially succeeded, achieving speed improvements of 25.6% for file downloads and 13.2% for HLS streaming for the attacker, compared to baseline speed. However, the implemented defense system proved highly effective, as it blocked the malicious optimistic ack packets. Multilevel defense system demonstrated robust protection, highlighting the system's ability to mitigate such attacks. Ongoing monitoring is recommended.