Misbehaving TCP Receivers Can Cause Internet-Wide Congestion Collapse

Rob Sherwood  Bobby Bhattacharjee  Ryan Braud

University of Maryland  UCSD
DoS Using TCP Congestion Control

- Sender transmits packet

Time

Sender

Receiver

1:1461
DoS Using TCP Congestion Control

- Sender transmits packet
- Receiver ACKs

Misbehaving TCP Receivers Can Cause Internet-Wide Congestion Collapse – p.2
DoS Using TCP Congestion Control

- Sender transmits packet
- Receiver ACKs
- Congestion window grows as long as ACKs continue

Sender transmits packet
Receiver ACKs
Congestion window grows as long as ACKs continue
DoS Using TCP Congestion Control

- Sender transmits packet
- Receiver ACKs
- Congestion window grows as long as ACKs continue
- ACKs are cumulative
DoS Using TCP Congestion Control

- From ACKs, sender infers:
  - Packet Loss
From ACKs, sender infers:
- Packet Loss
- When to send
- Data is buffered
DoS Using TCP Congestion Control

From ACKs, sender infers:
- Packet Loss
- When to send
- Data is buffered
- How much to send
- ACKs increase cwnd
DoS Using TCP Congestion Control

From ACKs, sender infers:
- Packet Loss
- When to send
- Data is buffered
- How much to send
- ACKs increase cwnd
- When to time out
- RTT estimation
DoS Using TCP Congestion Control

- From ACKs, sender infers:
  - Packet Loss
  - When to send
  - Data is buffered
  - How much to send
  - ACKs increase cwnd
  - When to time out
  - RTT estimation

- Assumes honest feedback
DoS Using TCP Congestion Control

From ACKs, sender infers:
- Packet Loss
- When to send
- Data is buffered
- How much to send
- ACKs increase cwnd
- When to time out
- RTT estimation

What if receiver is malicious?
DoS Using TCP Congestion Control

- Misbehaving receiver can send optimistic ACKs for packets before they are received
- Faster performance [Savage99]
Malicious receiver can ACK data that is never received

Force sender to congest the network (DoS)
Amplification

- Measure of DoS severity

\[ \text{Amplification} = \frac{\text{traffic generated}}{\text{traffic sent}} \]

- Flooding : x1
- Spoofed DNS: x4-x10
- Smurf (broadcast ping): x255
Amplification

- Measure of DoS severity

Amplification = \( \frac{\text{traffic generated}}{\text{traffic sent}} \)

- Flooding: \( \times 1 \)
- Spoofed DNS: \( \times 4 - \times 10 \)
- Smurf (broadcast ping): \( \times 255 \)
- OptAck: \( \times 1683 \)

- With window scaling: \( \times 32 \times 10^6 \)
**Talk Outline**

- **Contributions**
  - Discuss attack and amplification
  - Implementation techniques
    - Simulated and real world experiments
  - Solution: randomly skipped segments
  - Validate efficiency of implementation

- **Conclusions**
Attack Overview

Misbehaving TCP Receivers Can Cause Internet-Wide Congestion Collapse – p.7
Attack Overview

OptAcks

Incoming Data

Misbehaving TCP Receivers Can Cause Internet-Wide Congestion Collapse – p.7
Attack Overview

Network Wide DoS
Amplification Factor Analysis

\[
\begin{align*}
\text{traffic generated} & \quad \text{traffic sent} \\
\end{align*}
\]
Amplification Factor Analysis

\[
\frac{\text{# packets} \times \text{packet size}}{1 \ \text{ACK}}
\]

OptAck:
Send 1 ACK → Get entire window of packets
Amplification Factor Analysis

\[
\frac{cwnd}{mss} \times \left( \frac{40 + mss}{40} \right)
\]

- **cwnd**: Size of window
- **mss**: Packet payload size
- **40 bytes**: TCP header size
Amplification Factor Analysis

cwnd \times \left[ \frac{1}{\text{mss}} + \frac{1}{40} \right]

Typical: cwnd=2^{16}, mss=1460 \rightarrow x1683
Amplification Factor Analysis

\[ \text{cwnd} = \left(1 - \frac{1}{\text{mss}} + \frac{1}{40}\right) \]

- Typical: \( \text{cwnd} = 2^{16}, \text{mss} = 1460 \rightarrow x_{1683} \)

- [RFC1323] \( \text{cwnd} = 2^{16} \times 2^{\text{wscale}} \)
  - \( \text{wscale} = 14 \rightarrow \text{cwnd} = 2^{30} \)
Amplification Factor Analysis

\[ \text{cwnd} \times \left[ \frac{1}{\text{mss}} + \frac{1}{40} \right] \]

- Typical: cwnd=2^{16}, mss=1460 \rightarrow x1683
- [RFC1323] cwnd=2^{16} \times 2^{\text{wscale}}
  - wscale=14 \rightarrow \text{cwnd}=2^{30}
- Smaller mss \rightarrow more amplification
Amplification Factor Analysis

\[ \text{cwnd} \times \left[ \frac{1}{\text{mss}} + \frac{1}{40} \right] \]

- Typical: \( \text{cwnd}=2^{16}, \text{mss}=1460 \rightarrow x1683 \)
- [RFC1323] \( \text{cwnd}=2^{16} \times 2^{\text{wscale}} \)
  - \( \text{wscale}=14 \rightarrow \text{cwnd}=2^{30} \)
- Smaller mss \( \rightarrow \) more amplification
- \( \text{wscale}=14, \text{mss}=88 \rightarrow x32 \times 10^6 \)
Distributed Attack

- Target CDN’s and P2P File transfer nodes
Distributed Attack

- Target CDN’s and P2P File transfer nodes
- Use zombies to attack the Internet’s core
Distributed Attack

- Target CDN’s and P2P File transfer nodes
- Use zombies to attack the Internet’s core
  - Botnets of 100k+ nodes exist

[HONEYNET04]

How much traffic before collapse?

Open problem

Data point: Slammer worm generated 31GB/s at peak

Equal to traffic generated by 5 OptAttack attackers on T3s, wscale=0, mss=1460
Distributed Attack

- Target CDN’s and P2P File transfer nodes
- Use zombies to attack the Internet’s core
  - Botnets of 100k+ nodes exist
    [HONEYNET04]
- How much traffic before collapse?
Distributed Attack

- Target CDN’s and P2P File transfer nodes
- Use zombies to attack the Internet’s core
  - Botnets of 100k+ nodes exist
    - [HONEYNET04]
- How much traffic before collapse?
  - Open problem
Distributed Attack

- Target CDN’s and P2P File transfer nodes
- Use zombies to attack the Internet’s core
  - Botnets of 100k+ nodes exist [HONEYNET04]
- How much traffic before collapse?
  - Open problem
  - Data point: Slammer worm generated 31GB/s at peak [CAIDA03]
Target CDN’s and P2P File transfer nodes

Use zombies to attack the Internet’s core

Botnets of 100k+ nodes exist

How much traffic before collapse?

Open problem

Data point: Slammer worm generated 31GB/s at peak

Equal to traffic generated by 5 OptAck attackers on T3s, wscale=0, mss=1460
Is attack practical?

How vulnerable are deployed TCP Stacks?

Discovered effective solution

We implement the attack
  approx. 1200 lines of code

Main challenge: **ACK overruns**
ACK overruns

- Attacker becomes unsynchronized

![Diagram showing ACK overruns]

**Causes of Overrun**
- Server delay
- ACK compression
- Dropped ACKs

[RFC793] says ignore overrun ACKs. DoS otherwise.

Overrun avoidance and recovery.
ACK overruns

- Attacker becomes unsynchronized

---

**Diagram:**
- Sender ( Victim )
- Attacker
- ACK Arrives Early
- OS Delay
- ACK 14601
- 14601:16061
- 16061:17521
- 17521:18981
- 18981:20441
- ACK 21901
- 20441:21901

---

Misbehaving TCP Receivers Can Cause Internet-Wide Congestion Collapse – p.11
ACK overruns

- Attacker becomes unsynchronized

Causes of Overrun:
- Server delay
- ACK compression
- Dropped ACKs

[RFC793] says ignore overrun ACKs. DoS otherwise.

Overrun avoidance and recovery.
ACK overruns

- Attacker becomes unsynchronized

Causes of Overrun:
- Server delay
- ACK compression
- Dropped ACKs

[RFC793] says ignore overrun ACKs or DoS otherwise.

Overrun avoidance and recovery:

OS Delay

Retransmit after RTO

Attacker Continues Blindly

ACK Arrives Early

Reception by Sender (Victim):
- T1 = 14601
- T2 = 21901
- T3 = 30661
- T4 = 40441
- T5 = 50221
- T6 = 60001

Retransmit after RTO
ACK overruns

- Attacker becomes unsynchronized
- Causes of Overrun
  - Server delay
  - ACK compression
  - Dropped ACKs

Overrun Avoidance and Recovery

[Diagram showing time line with various time stamps and interactions between sender and attacker, illustrating the process of overruns and recovery]

[RFC 793] says ignore overrun ACKs, otherwise DoS.
ACK overruns

- Attacker becomes unsynchronized
- Causes of Overrun
  - Server delay
  - ACK compression
  - Dropped ACKs
- [RFC793] says ignore overrun ACKs
  - DoS otherwise
ACK overruns

- Attacker becomes unsynchronized
- Causes of Overrun
  - Server delay
  - ACK compression
  - Dropped ACKs
- [RFC793] says ignore overrun ACKs
  - DoS otherwise
- Overrun avoidance and recovery
Overrun Avoidance and Recovery

- Server delays are **short** (typically <50ms)
Overrun Avoidance and Recovery

- Server delays are short (typically <50ms)
- Attacker should target more servers
Overrun Avoidance and Recovery

- Server delays are short (typically <50ms)
  - Attacker should target more servers

- Don’t ACK whole window
  - ACK $\frac{1}{2} \textit{cwnd}$, twice as often
  - Thwarts ACK compression, dropped ACKs
Overrun Avoidance and Recovery

- Server delays are short (typically <50ms)
- Attacker should target more servers
- Don’t ACK whole window
  - ACK $\frac{1}{2}$ cwnd, twice as often
  - Thwarts ACK compression, dropped ACKs
- Don’t exceed local attack bandwidth
Overrun Avoidance and Recovery

- Server delays are short (typically <50ms)
- Attacker should target more servers
- Don’t ACK whole window
  - ACK $\frac{1}{2} \text{cwnd}$, twice as often
  - Thwarts ACK compression, dropped ACKs
- Don’t exceed local attack bandwidth
- Recovery details in paper
Experiment: How much traffic does OptAck generate?
Connections:
1 Attacker: 1.5Mbps
Victims: 100Mbps

Parameters:
wscale=4
mss=1460
Connections:
1 Attacker: 1.5Mbps
Victims: 100Mbps

Max traffic:
99.9% of predicted

Conclusion:
Model is accurate
1 Attacker on LAN w/ 1 victim
wscale=0, mss=1460
No multi-victim overrun avoidance
Results - Implementation

- 1 Attacker on LAN w/ 1 victim
- wscale=0, mss=1460
- **Conclusion:** Works on deployed TCP stacks

![Graph showing Amplification Factor for different operating systems](image-url)
Defenses

How to defend against OptAck?
Defenses

How to defend against OptAck?

- Nonce
  - Receiver modification; not deployable
Defenses

How to defend against OptAck?

- Nonce
  - Receiver modification; not deployable
- Rate limiting
  - Target more servers; use more attackers
Defenses

How to defend against OptAck?

- Nonce
  - Receiver modification; not deployable
- Rate limiting
  - Target more servers; use more attackers
- RST on overrun ACK
  - Trivial DoS
Defenses

How to defend against OptAck?

- Nonce
  - Receiver modification; not deployable
- Rate limiting
  - Target more servers; use more attackers
- RST on overrun ACK
  - Trivial DoS
- Network Policing
Defenses

How to defend against OptAck?

- Nonce
  - Receiver modification; not deployable
- Rate limiting
  - Target more servers; use more attackers
- RST on overrun ACK
  - Trivial DoS
- Network Policing

**Insight:** attacker cannot tell where in the network a packet is dropped
Defenses: Skipped Segments

Sender should randomly, periodically, skip sending a data segment.
Defense: Skipped Segments

- Sender should randomly, periodically, skip sending a data segment
- Good receiver sends duplicate ACKs; fast retransmit skipped segment
- Malicious receiver ACKs the hole; RST connection
Defense: Skipped Segments

- Sender should randomly, periodically, skip sending a data segment
  - Good receiver sends duplicate ACKs; fast retransmit skipped segment
  - Malicious receiver ACKs the hole; RST connection

- Cost:
  - Transmission rate unchanged
  - Application receives data 1 RTT later
Defense: Skipped Segments

- Sender should randomly, periodically, skip sending a data segment
- Good receiver sends duplicate ACKs; fast retransmit skipped segment
- Malicious receiver ACKs the hole; RST connection

Cost:
- Transmission rate unchanged
- Application receives data 1 RTT later

Sender only modification: deployable
Defense: Skipped Segment (More)

- Sender should not back off
Defense: Skipped Segment (More)

- Sender should not back off
- Cost of RNG too high per packet
  - Init counter randomly from [low, high]
  - Decrement counter with each segment
  - counter==0 → skip segment
  - re’init counter randomly
Defense: Skipped Segment (More)

- Sender should **not** back off
- Cost of RNG too high per packet
  - Init counter *randomly* from \([low, high]\)
  - Decrement counter with each segment
  - counter==0 → skip segment
  - re’init counter randomly
- TCP optimized to recover from packet loss
Defense: Skipped Segment (More)

- Sender should not back off
- Cost of RNG too high per packet
  - Init counter randomly from \([low, high]\)
  - Decrement counter with each segment
  - counter==0 → skip segment
  - re’init counter randomly
- TCP optimized to recover from packet loss
- Linux implementation:
  - 30 lines of code
  - 5 bytes per connection
Skipped Segments Efficiency

- Download 100MB file
- No attack
- Vary skip rate

**Conclusion:**
Overhead is trivial < 0.1%.
Skipped Segments Efficiency

- Download 100MB file
- No attack
- Vary skip rate

**Conclusion:**
Overhead is trivial

< 0.1%

Percent Overhead

95% Confidence Interval

Average

Misbehaving TCP Receivers Can Cause Internet-Wide Congestion Collapse – p.19
Related Work

[Savage99] TCP Congestion Control with a Misbehaving Receiver.
Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson.
CCR 1999.

- Original optimistic acknowledgment
- Defense requires receiver side modifications - inhibits deployment
draft-azcorra-tcpm-tcp-blind-ack-dos-01

Proposed solutions already discussed in paper
Related Work (more)


Conclusion

- OptAck attack is dangerous: \( x_{1683} \)
- \( x_{32} \times 10^6 \text{ w/ wscaling} \)
Conclusion

- OptAck attack is dangerous: $1683$
  - $32 \times 10^6$ w/ wscaling
- Deployed TCP Stacks are vulnerable
Conclusion

- OptAck attack is dangerous: $x^{1683}$
  - $x^{32} \times 10^6$ w/ wscaling
- Deployed TCP Stacks are vulnerable
- Skipped segments solution:
  - performs efficiently
  - requires sender only deployment
Conclusion

- OptAck attack is dangerous: $1683 \times 32 \times 10^6$ w/ wscaling
- Deployed TCP Stacks are vulnerable
- Skipped segments solution:
  - performs efficiently
  - requires sender only deployment
- Future work: DETER?
Conclusion

- OptAck attack is dangerous: $x^{1683}$
  - $x^{32 \times 10^6}$ w/ w.scaling
- Deployed TCP Stacks are vulnerable
- Skipped segments solution:
  - performs efficiently
  - requires sender only deployment
- Future work: DETER?
- Questions?

This talk was written in \textsc{latex} with Prosper:
http://prosper.sourceforge.net/
Defense Categories

- bandwidth caps
  - just use more victims or more attackers
- nonces
  - non-cumulative nonces are not robust
  - cumulative nonces require client deployment
- ACK alignment/timestamps
  - requires CPU for strong random numbers
  - timestamps are optional
  - not robust to lazy attack
More Defenses

- In network support
  - not currently deployed
- Random Pauses
  - inefficient
- segment reordering
  - allows false positives from network reordering, dropped ACKs
  - no performance penalty