Protocol Attacks

By
Sushant Rewaskar

Outline : Part 1

- Introduction
- What is a “protocol attack”?
- How does it work?
- Different types of protocol attack

Introduction: Types of attacks

- Buffer overflow
- Weak authentication/encryption
- Inadequate argument checking
- Configuration errors
- Insecure program features
- Kernel-level problems
- Protocol attack

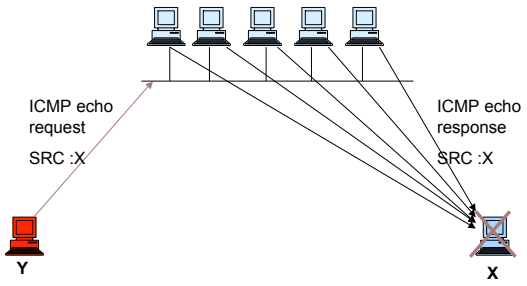
What is a protocol attack?

- Exploit a specific feature or implementation bug of some protocol installed at the victim in order to consume excess amounts of its resources

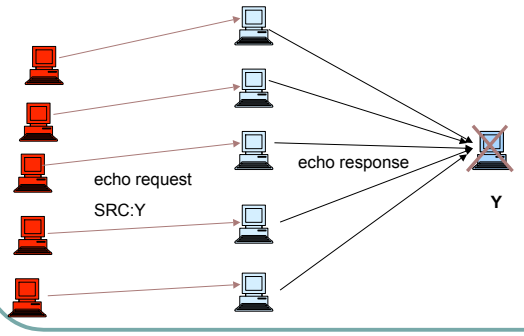
Popular Protocol attack

- Smurf Attack
- SYN attack
- UDP Attack, ICMP Attack
- CGI request attack
- Authentication server attack
- Attack using DNS systems.
- Attack using spoofed address in ping

Smurf Attack



UDP Attack, ICMP Attack, Ping attack

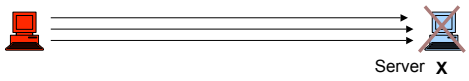


TCP SYN

- Uses TCP's 3 way hand shake
- Send a SYN packet with a spoofed IP address
- Server is not able to complete the handshake and as a result wastes all its network resources

CGI request attack

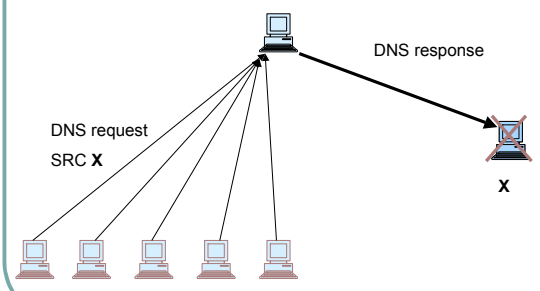
- CGI script uses CPU cycles to satisfy a request.
- Attacker send multiple CGI requests
- This consumes precious CPU cycle on the server



Authentication server attack

- Authentication server validates a signature
- It takes more resources to check a bogus signature then to create it.
- Attacker send a bogus signature to the server

Attack using DNS systems.



Feature of these attacks

- All attacks need a lot of attackers (zombies)
- Mitigate by changing the protocol features
- Line between protocol and brute force commands is very thin
- Can these attacks be identified?
  - YES

Conclusion : Part 1

- High-Rate Protocol attack
  - Very close to Brute force attack

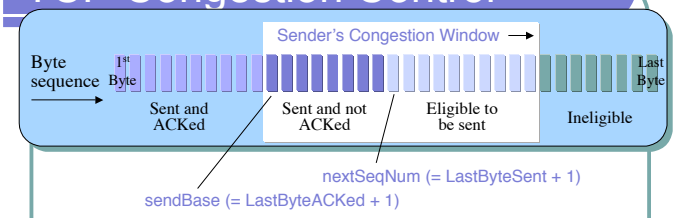
Alternate Protocol attacks

- Use some feature of the protocol to launch an attack without being aggressive
- Can this be done?
  - Yes
    - Misbehaving receiver attack
    - Shrew attack

Outline : Part 2

- TCP mechanism
  - Congestion window modification
  - Congestion avoidance
- Design attack to make use of congestion window update on acks
- Evaluate attack's efficiency
- TCP modification to prevent the attack

TCP Congestion Control



- Transmission rate is limited by the *congestion window* size, `congWin`

  `LastByteSent - LastByteACKed ≤ MIN(congWin,RcvWindow)`
- Maximum rate is *w* MSS byte segments sent every RTT

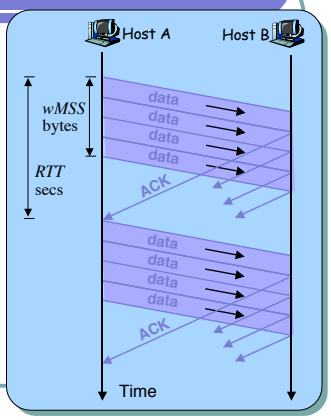
$$throughput = \frac{w \times MSS}{RTT} \quad bytes/sec$$

TCP Congestion Control

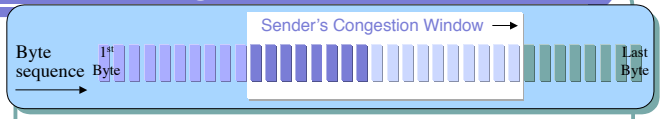
- If $w \times MSS/R < RTT$, then the maximum rate at which a TCP connection can transmit data is

$$\frac{w \times MSS}{RTT} \quad bytes/sec$$

- *w* is the minimum of the number of segments in the receiver's window or the congestion window



TCP Congestion Control



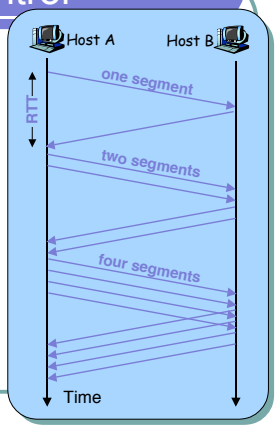
- TCP connections probe for available bandwidth
  - Increase the congestion window until loss occurs
  - When loss is detected decrease window, then begin probing (increasing) again
- The congestion window grows in two phases:
  - *Slow start* — Ramp up transmission rate until loss occurs
  - *Congestion avoidance* — Keep connection close to sustainable bandwidth
- A window size threshold (bytes transmitted) distinguishes between slow start and congestion avoidance phases

TCP Congestion Control

```
congWin = 1 MSS
for (each original ACK received)  congWin++
   until (loss event OR congWin > threshold)
```

- Exponential increase in window size each RTT until:
  - Loss occurs
  - *congWin = threshold*

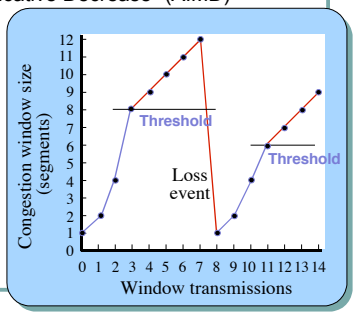
  (Not so slow!)
- Note: TCP implementations detect loss differently
  - TCP "Tahoe": Timeout
  - TCP " Reno": Timeout or three duplicate ACKs



TCP Congestion Control

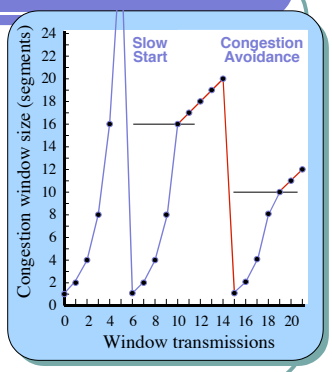
- Increase congestion window by 1 segment each RTT, decrease by a factor of 2 when packet loss is detected
  - "Additive Increase, Multiplicative Decrease" (AIMD)

```
/* slowstart is over;
   congWin > threshold
*/
until (loss event) {
  whenever congWin segments
     ACKed:
       congWin++
  }
/* loss event timeout */
threshold = congWin/2
congWin  = 1 MSS
perform slowstart
```



TCP Congestion Control

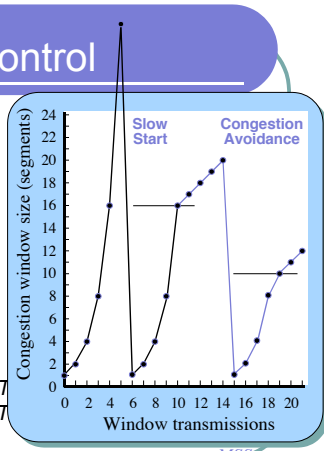
- The threshold is an estimate of a "safe" level of throughput that is sustainable in the network
  - The threshold specifies a throughput that was sustainable in the recent past
- Slow-start quickly increases throughput to this threshold
- Congestion avoidance slows probes for additional available bandwidth beyond the threshold



Assume $RTT > \frac{w \times MSS}{R}$

TCP Congestion Control

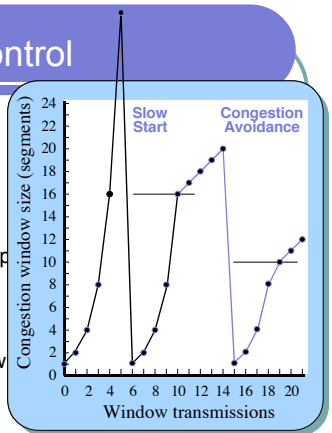
- Loss (at any time) reduces the "safe" throughput estimate to 1/2 of the current throughput
  - This is the throughput that resulted in loss
- Slow-start begins anew whenever there is loss
- Throughput at initial threshold = 1 MB/*RTT*
  - At 1st threshold: 16*MSS*/*RTT*
  - At 2nd threshold: 10*MSS*/*RTT*



Assume $RTT > \frac{w \times MSS}{R}$

TCP Congestion Control

- TCP Tahoe:
  - Loss signaled by timeout
  - *threshold = congWin*/2
  - *congWin* = 1 MSS
- TCP Reno:
  - "Fast retransmit" — Receip of 3 duplicate ACKs also signals a packet loss
  - "Fast recovery" — Skips slowstart and continue in congestion avoidance new slowstart threshold
- Others: TCP NewReno, SACK, …



Assume $RTT > \frac{w \times MSS}{R}$

Outline : Part 2

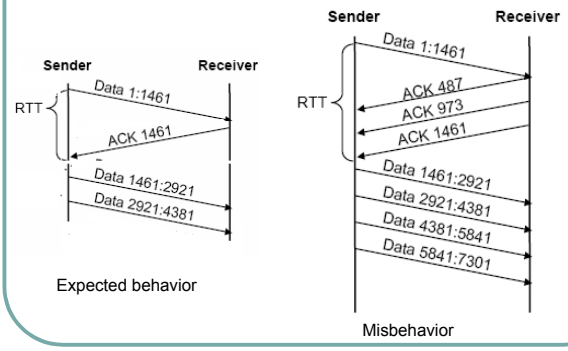
- TCP mechanism
  - Congestion window modification
  - Congestion avoidance
- Design attack to make use of congestion window update on acks
- Evaluate attack's efficiency
- TCP modification to prevent the attack

TCP Mechanism

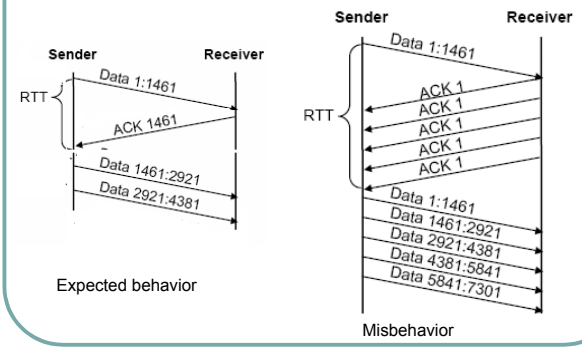
- Tcp work at two granularities
  - Acks received and processed at bytes granularity
  - Updates to window performed at packet granularity

A clever receiver can use this to its benefit

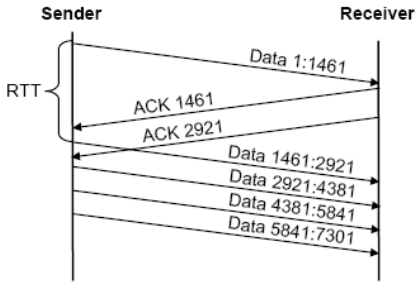
Ack division



DupAck spoofing



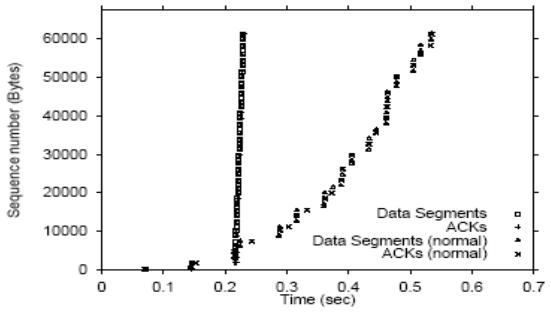
Optimistic Acking



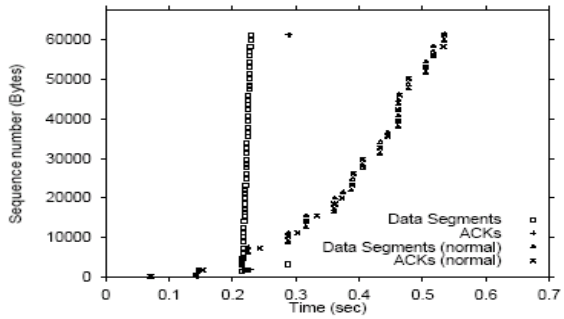
Outline : Part 2

- TCP mechanism
  - Congestion window modification
  - Congestion avoidance
- Design attack to make use of congestion window update on acks
- Evaluate attack's efficiency
- TCP modification to prevent the attack

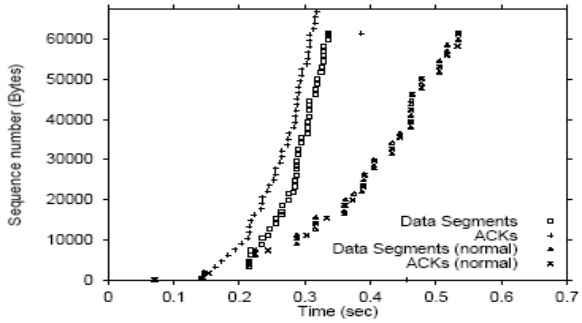
Evaluation: Ack division



Evaluation : Ack spoofing



Evaluation : Optimistic acking



Outline : Part 2

- TCP mechanism
  - Congestion window modification
  - Congestion avoidance
- Design attack to make use of congestion window update on acks
- Evaluate attack's efficiency
- TCP modification to prevent the attack

Solution

- Ack division
  - Increment congestion window only when you get MSS bytes of data
- DupAck spoofing
  - Use a Nonce
- Optimistic Acking
  - Sum of Nonce in the acks

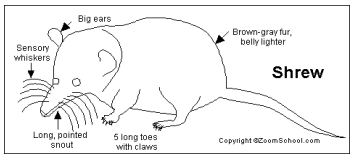
Conclusion Part 2

- Features of a Protocol can be used to modify its behavior in a harmful way.

Part 3 : Outline

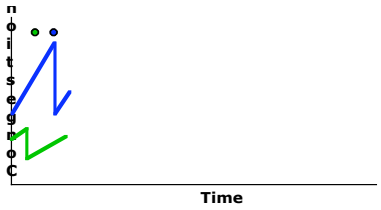
- Design attack to take advantage of the congestion avoidance mechanism (shrew attack)
- Explore TCP's response to shrew attack
  - Modeling, simulation, Internet experiments
- Evaluate detection mechanism

Shrew



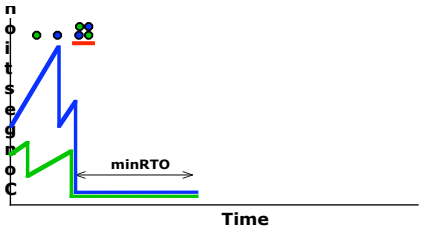
- Very small but aggressive mammal that ferociously attacks and kills much larger animals with a venomous bite

TCP mechanism- AIMD



- Operates at the RTT time-scale

TCP mechanism- timeout



- Operates at the RTO timer time-scale
  - seconds

TCP dual time scale operation

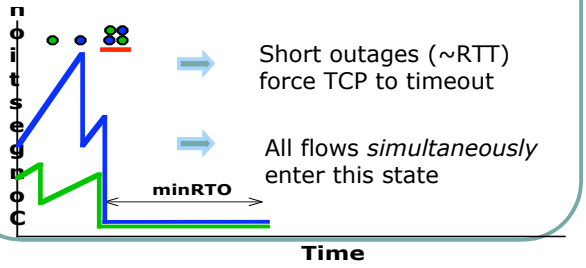
- TCP operates at two time-scales
  - RTT time-scales (~10-100 ms)
    - AIMD control
  - RTO time-scales (*RTO=SRTT+4*RTTVAR*)
    - Avoid congestion collapse
- RTO must be lower bounded to avoid spurious retransmissions
  - [AllPax99] and RFC2988 recommends minRTO = 1 sec

Outline : Part 3

- Analyze TCP congestion avoidance
- Design attack to take advantage of the mechanism (shrew attack)
- Explore TCP response to shrew attack
  - Modeling, simulation, Internet experiments
- Evaluate detection mechanism

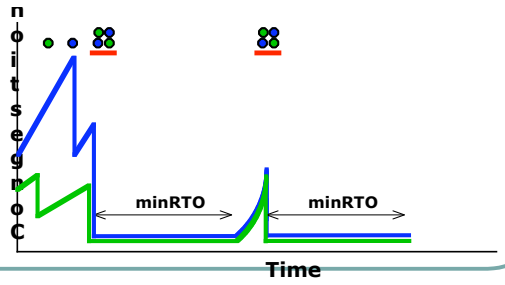
Shrew Attack

- Pulse induced outages
  - create along enough outage so that all flows experience a loss

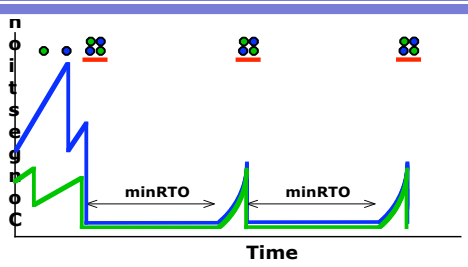


Shrew Attack

- Induce an outage again after minRTO



Shrew Attack

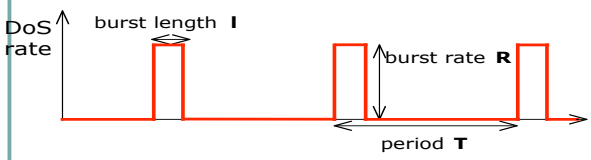


- Shrew periodically repeats pulse
  - RTT-time-scale outages inter-spaced on minRTO periods can deny service to TCP

Principles of Shrew

- Shrews exploit protocol homogeneity and determinism
  - Protocols react in a pre-defined way
  - Tradeoff of vulnerability vs. predictability
- Periodic outages synchronize TCP flow states and deny their service
- Slow time scale protocol mechanisms enable low-rate attacks
  - Outages at RTO scale, pulses at RTT scale imply low average rate

Creating periodic outage

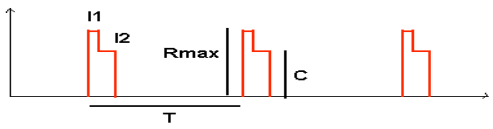


- Let
  - l = l1+l2
    - l1 = time to fill the queue
    - l2 = RTT time-scale
  - T = minRTO time-scale

Aggressiveness of stream

- Average rate of the stream
  - Avg = (l * R) /T
  - l = 10-100 ms
  - T= 1 second
  - R >=link capacity
- Average rate is ~1/10th of the link capacity

Optimized stream



- Average rate
  - (l1*R + l2*C) /T

Analytical model for shrew

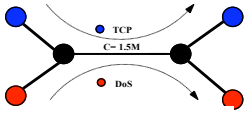
- Consider a periodic DOS stream. Let, outage duration satisfy following two conditions
  - L>RTT
  - minRTO >SRTT+4*RTTVAR
- Throughput is given by
  - $$\rho(T) = \frac{ceil(\frac{\min RTO}{T}) * T - \min RTO}{T}$$

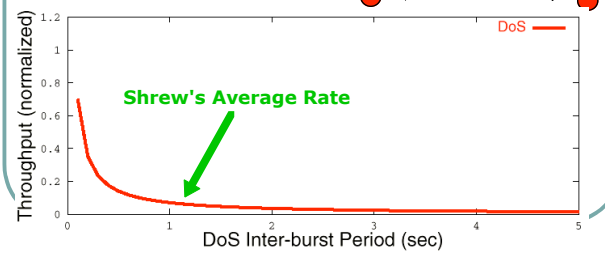
Outline : Part 3

- Analyze TCP congestion avoidance
- Design attack to take advantage of the mechanism (shrew attack)
- Explore TCP response to shrew attack
  - Modeling, simulation, Internet experiments
- Evaluate detection mechanism

Shrew : DOS rate

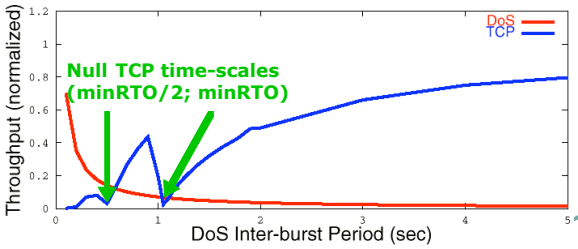
- Let C=R=1.5Mbps
- L = 70ms





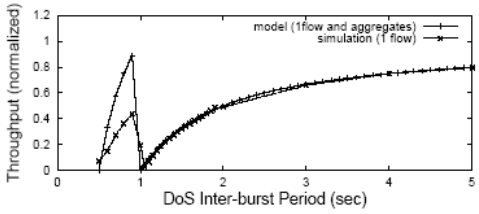
Shrew: model

- Analytical model predicts the outage at minRTO



Shrew :Simulation

- Ns simulation with l = 150ms, RTT = 12-132ms, C= 1.5ms and minRTO =1 s

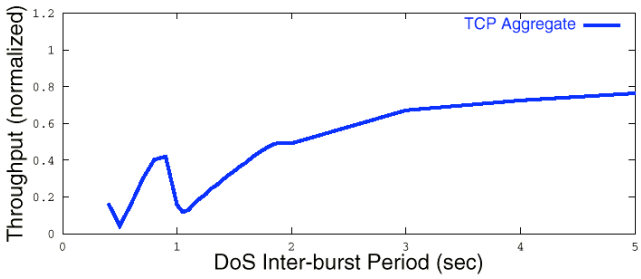


Challenge for shrew

- **Aggregation**
  - Vulnerable due to Shrew-induced flow synchronization
- **RTT heterogeneity**
  - Shrews are high-RTT pass filters
- **DoS peak rate**
  - Less-than-bottleneck bursts can damage short-RTT flows
- **TCP Variants**
  - For short burst length, TCP Reno is more fragile
- **Short-lived TCP flows**
  - Web browsing
- **Internet experiments**
  - Can Shrews be successful on the Internet?

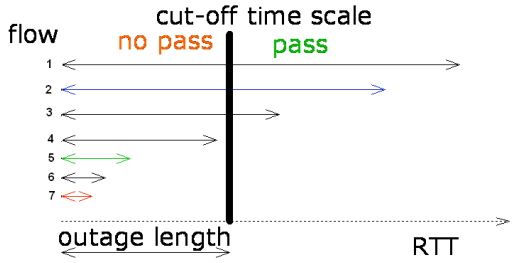
Aggregation

- TCP flows with homogeneous RTT are vulnerable
- Shrews induce flow synchronization



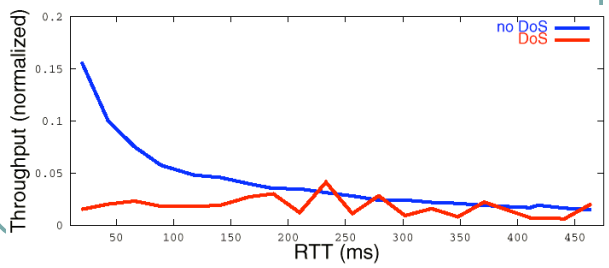
RTT heterogeneity

- Shrew acts as a high-RTT pass filter



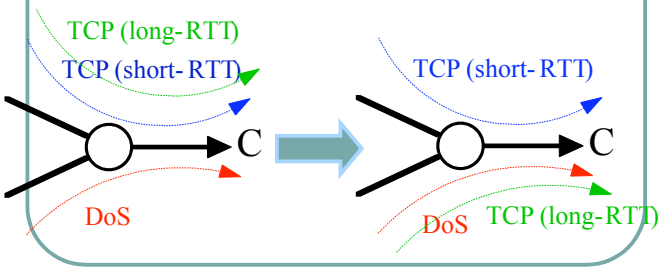
RTT heterogeneity

- 20 TCP flows; RTT ~ 20-460ms
- Cut off timescale ~180ms



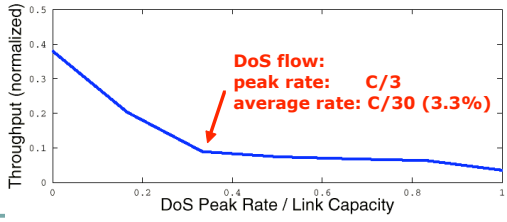
DoS peak rate

- Long flows collaborate in the attack



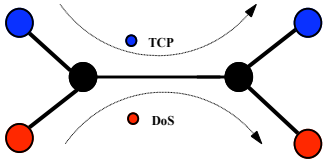
DoS peak rate

- Scenario: 4 TCP flows + DoS
  - 1 short-RTT & 3 long-RTT flows
  - DoS outage ~ RTT of the short-RTT flow



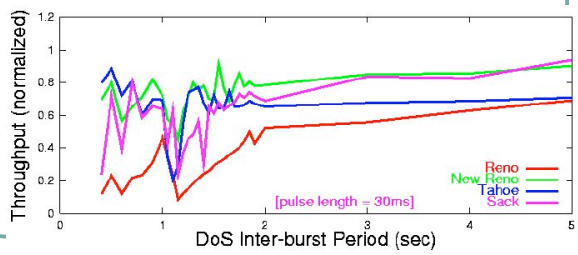
TCP variants

- So far we have seen TCP reno
- What about newReno, SACK?
- Experiment
  - TCP variant
    - Reno
    - New Reno
    - Tahoe
    - SACK
  - DoS stream
    - Burst rate = bottleneck capacity.
    - Burst length: 30ms, 50ms, 70ms, and 90ms



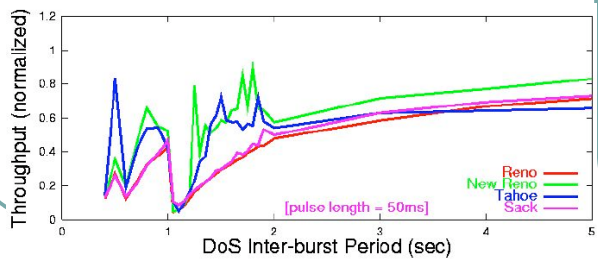
TCP variants

- Burst length = 30ms
  - TCP Reno is most fragile



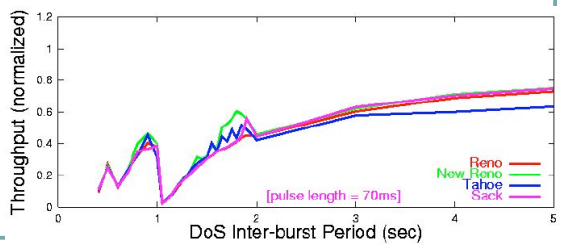
TCP variants

- Burst length = 50ms
  - TCP is the most vulnerable in 1-1.2 sec time-scale region due to slow start



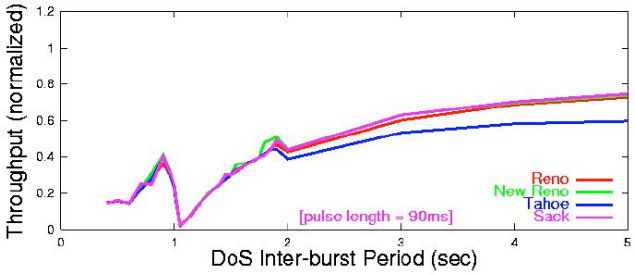
TCP variants

- Burst length = 70ms
  - Sufficient pulse width ensures timeout



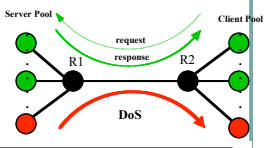
TCP variants

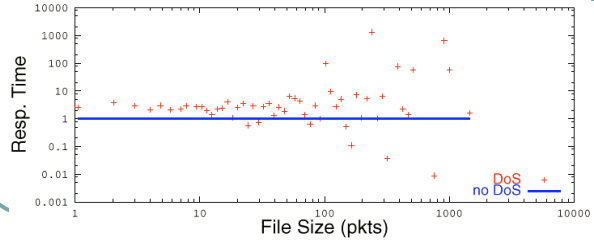
- Burst length = 90ms
  - With large burst length all TCP are equally fragile



Short-lived TCP flows

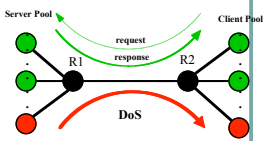
- Scenario: Web browsing (50% load)
  - Average damage to
    - a mouse (<100pkts) **=400%** delay increase
    - an elephant (>100pkts) **=24500%** delay increase

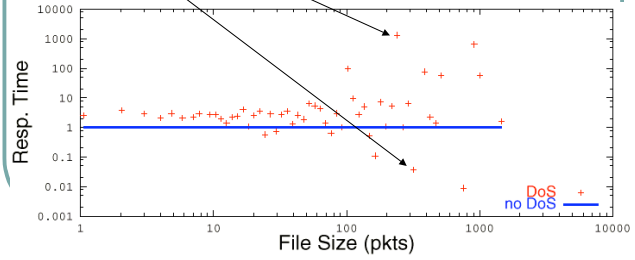




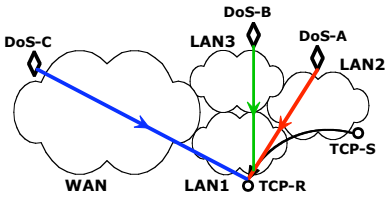
Short-lived TCP flows

- Scenario: Web browsing (50% load)
  - **Larger files more vulnerable**
    - most suffer
    - some benefit





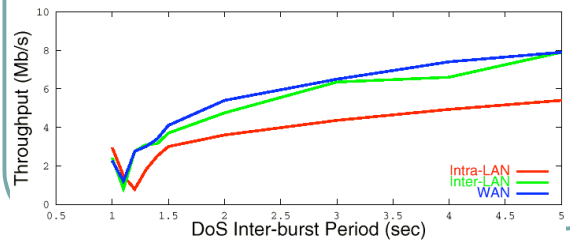
Internet experiments



- Scenario: victim on a lightly loaded 10 Mb/sec LAN
- Attacker on same LAN, nearby LAN, or over WAN
- WAN path:
  - EPFL→ETH, 8 hops (10/100/OC-12)

Internet experiments

- Shrew average rate: 909 kb/sec
  - R = 10 Mb/sec, l = 100 msec, T = 1.1 sec
- TCP throughput
  - 9.8 Mb/sec without Shrew
  - 1.2 Mb/sec with Shrew, 87.8% degradation



Outline : Part 3

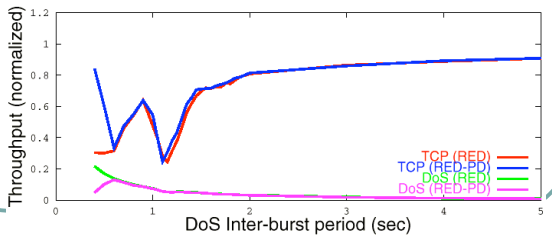
- Analyze TCP congestion avoidance
- Design attack to take advantage of the mechanism (shrew attack)
- Explore TCP response to shrew attack
  - Modeling, simulation, Internet experiments
- Evaluate detection mechanism

Detecting Shrews

- Shrews have low average rate, yet send high-rate bursts on short time-scales
- **Key questions**
  - Can algorithms intended to find high-rate attacks detect Shrews?
  - Can we tune the algorithms to detect Shrews without having too many false alarms?
- A number of schemes can detect malicious flows
  - E.g., RED-PD:
    - use the packet drop history to detect high-bandwidth flows and preferentially drop packets from these flows

Router-Assisted Mechanisms

- Scenario: 9 TCP Sack flows with RED and RED-PD
- RED-PD only detects Shrews with unnecessarily high rate
- Reducing RED-PD measurement time scale results in excessive false positives



End-point minRTO Randomization

- Observe
  - Shrews exploit protocol homogeneity and determinism
- Question
  - **Can *minRTO* randomization alleviate threat of Shrews?**
- TCP flows' approach
  - Randomize the *minRTO* = uniform*(a,b)*
- Shrews' counter approach
  - Given flows randomize minRTO, the optimal Shrew pulses at time-scale **T=b**
    - Wait for all flows to recover and then pulse again

End-point minRTO Randomization

- TCP throughput for T=b time-scale of the Shrew attack

$$\rho(T=b) = \frac{n}{n+1}\frac{b-a}{b}$$

n - number of TCP flows
a,b - param. of unif. dist.

  - ***a*** small → spurious retransmissions [AllPax99]
  - ***b*** large → bad for short-lived (HTTP) traffic

- *Randomizing the minRTO* parameter shifts and smoothes TCP's null time-scales
- **Fundamental tradeoff** between TCP performance and vulnerability to low-rate DoS attacks remains

Conclusions : Part 3

- Shrew principles
  - Exploit slow-time-scale protocol homogeneity and determinism
- Real-world vulnerability to Shrew attacks
  - Internet experiment: 87.8% throughput loss without detection
- Shrews are difficult to detect
  - Low average rate and "TCP friendly"
  - Cannot filter short bursts
  - Fundamental mismatch of attack/defense timescales

Open Questions

- Can filters specific to Shrews be designed without excessive false positives?
- Can end-point algorithms be sufficiently randomized, so that
  - attackers cannot exploit their known reactions
  - performance is not sacrificed