An API for dynamic firewall control and its implementation for Linux Netfilter

3. Essener Workshop
"Neue Herausforderungen in der Netzsicherheit"

Jochen Kögel, Sebastian Kiesel, Sebastian Meier
jochen.koegel@ikr.uni-stuttgart.de
4. April 2008

Universität Stuttgart
Institute of Communication Networks and Computer Engineering (IKR)
Prof. Dr.-Ing. Dr. h.c. mult. P. J. Kühn
Agenda

• Problem statement
• API requirements and design
• Implementation for Netfilter
• Performance evaluation: measurement results
• Possible improvements
• Conclusion and Outlook
Problem statement

Dynamic firewall control

Security at network edge: Open firewalls for legitimate connections

• for VoIP: SIP/SDP and RTP
  – strict policies – authorization of SIP sessions
  – open firewall (pinhole) for media stream, parameters negotiated with SIP/SDP
  – two firewall parts: signaling component and media component

• several approaches possible
  – distributed vs. monolithic (Session Border Controller – SBC)
  – packet filter vs. RTP proxy
Problem statement

Dynamic firewall control

Approach: distributed + packet filter (using firewall control protocols)

- server process running on firewall machines manages pinholes
- accepting only messages from authorized machines
- session stateful server (SIP B2BUA)
  - extracts RTP-flow parameters from signaling messages
  - authorizes calls
  - signals pinholes to open/close
- several controlling instances (e.g. also Intrusion Detection Systems)
Problem statement

Dynamic firewall control

Design of firewall control daemon @IKR (SIMCO server)

• how to open pinholes?
  - calling command line tools?
  - using libraries (libiptc, nfnetlink)?

• daemon runs on different Operating Systems, what about packet filter dependencies?
  → packet filter interface is very OS-specific (and even in Linux there are several)
  → general pinhole API, not only for SIMCO server
API design

Requirements from firewall control frameworks

MIDCOM/SIMCO

- implementation of Midcom: simple middlebox control protocol (SIMCO), (RFC 4540)
- NAT + packet filter signaling – our focus: packet filter
- enable (PER) and prohibit (PDR) pinholes (white list)
  → PDR closes affected pinholes (bulk change)
- pinhole
  - two "address tuples" (transport protocol, address, prefix, port, portrange)
  - ports and address wildcarding
  - inbound/outbound/bidirectional
→ pinhole: five tuple with ranges/prefix, white list

problem: multiple packet filters at network edge
  - must be handled by client, independent of packet filters
  - 1st possibility: know routing
  - 2nd possibility: open pinholes in every packet filter
API design

Requirements from firewall control frameworks

IETF NSIS (next steps in signaling)

- framework for path-coupled signaling
  - idea: signal nodes on path independent of IP routing (e.g. for QoS)
  - generic messaging layer (General Internet Signaling Transport)
    - Datagram/Connection Mode
    - TCP, UDP, IPSec
  - NSIS Signaling Level Protocols (NSLP) on top of GIST

- NAT/Firewall Control
  - NAT/Firewall control NSLP (draft-ietf-nsis-nslp-natfw-18.txt)
  - authorization possible with tokens (draft-manner-nsis-nslp-auth-03.txt)
**API design**

*Requirements from firewall control frameworks*

**IETF NSIS (next steps in signaling)**

- pinhole description based on NSIS-flow
  - `sub_ports`: number of contiguous ports (0..1)
  - typically white list approach for pinholes
  - also traffic blocking mode with EXT messages (for whole prefix, port wildcard)

→ pinhole as five tuple, range definitions are subset of simco
→ white list feasible. Blocking can be mapped to shrinking the white list
API design

Requirements from firewall control frameworks

Requirements

- open/close pinholes
- unidirectional pinhole: five tuple (incl. subnets + port ranges)
  - bidirectional: two pinholes
  - for TCP: direction of connection establishment
- independent of filter implementation (and OS)
- transaction semantics (typically, several rules are added at once)
- performance
  - frequent rule changes (VoIP)
  - high packet rate
- security
  - no control over whole packet filter, only dedicated rule sets
  - controlling entity must not be root, else a compromised firewall control daemon is fatal
API design

Managing pinholes using the pinhole API

features
• white list approach
• rules defined by five tuple
  + prefix length
  + port range
• adding rules by definition (returns ID)
• removing rules by ID

simple transaction mechanism
1. start transaction
2. add/delete rules
3. commit
API design

Implementation aspects

The big picture

- application design independent of operating system
- use of different packet filter by changing translation plugin
- use of different packet filters depending on rule type (optimization possible)
API design

Implementation aspects

Frontend

- keeps all rules/pinholes
  - optimization possible (hook) while still being able to delete rules per ID
  - enables differential updates
  - failure: last known good
- commit rules as batch to backend
- socket communication: reuse of SIMCO message definition + added new control messages
API design

Implementation aspects

Backend
- processing of frontend requests
- translation of pinholes to netfilter rules
- notify frontend about status
- failure recovery, e.g. frontend crash
- only Translation module II is packetfilter-dependent

Diagram:
- Unix Domain Socket Adapter
- Simco Lite Server
- Translation I
- Translation II
- Control
- Internal interface from Frontend

API functions:
- init()
- flush()
- add()
- insert()
- del()
- commit()
Implementation for Linux Netfilter

Mapping rules to Netfilter

Legend
- Netfilter hook
- packet + metadata

sockets/applications

IP Stack

PREROUTING

INPUT

FORWARD

OUTPUT

POSTROUTING
Implementation for Linux Netfilter

Mapping rules to Netfilter

Legend

- Netfilter hook
- packet + metadata

IP Stack
Netfilter Modules

Conntack
iptables/Filter
Ipset
Implementation for Linux Netfilter

Mapping rules to Netfilter

- **sockets/applications**
- **iptables/FilterConntrack Ipset**
- **Netfilter Modules**
  - **POSTROUTING**
  - **PREROUTING**
  - **INPUT**
  - **OUTPUT**
  - **FORWARD**

Legend:
- **Netfilter hook**
- **packet + metadata**
- **filter rule**
  - conditions (match)
  - action (target)

Legend:
- **Netfilter hook**
- **packet + metadata**
- **filter rule**
  - conditions (match)
  - action (target)
Implementation for Linux Netfilter

Mapping rules to Netfilter

Legend
- Netfilter hook
- packet + metadata
- filter rule
- conditions (match)
- action (target)

PREROUTING
 INPUT
 FORWARD
 OUTPUT
 POSTROUTING

sockets/applications

Netfilter Modules
- POSTROUTING
- OUTPUT
- INPUT
- PREROUTING

IP Stack

Legend
- Netfilter hook
- packet + metadata
- filter rule
- conditions (match)
- action (target)

not shown: details on NAT, mangle
Implementation for Linux Netfilter

Mapping rules to Netfilter

- iptables/Filter
- Ipset
- hashtables
- bitmaps
- trees

Legend:
- Netfilter hook
- packet + metadata
- filter rule
- conditions (match)
- action (target)

- PREROUTING
- INPUT
- FORWARD
- OUTPUT
- POSTROUTING

protocol helper

packet/flow classification based on hashtable

sysadmin’s rules
controlled by backend

not shown: details on NAT, mangle
Implementation for Linux Netfilter

Netfilter Modules

• iptables
  – linear search over lists (chains)
  – extensible by sophisticated "matches"

• connection tracking (contrack)
  – stateful packet filter
  – hash-based connection table
  – determines connection state and stores it to packet metadata

• ipset
  – hash-, tree- and bitmap based filter modules
  – realized as iptables march – stateless

• nf-HiPAC (High Performance PAcket Classification)
  – fast for high number of rules
  – possible replacement for chains/tables
  – patch for older kernels

→ pinhole API implemented for tables/chains, since port ranges and subnets required.
   (conntrack and ipset work for exact match only, nf-HiPAC is not integrated)
Impementation for Linux Netfilter

Managing netfilter rules

Accessing iptables – LibIPTC

- different representations in user and kernel space
- translation of complete ruleset before and after modifications
Performance Evaluation

Measurements with libiptc backend (VoIP Scenario)

Parameters

• 20 ms packetizing time: 100 pps/call (bidirectional), no bursts
• 2 pinholes per call: asymmetric RTP
→ rate and rule set depending on number of simultaneous calls
• Pentium 4, 2.53 GHz

Measurement Scenarios

• transaction delay for entering/Removing rules without network traffic
• packet loss and delay for traffic traversing the packet filter
  1. legitimate traffic only
  2. additionally with "bad" traffic, that will be filtered
    • contributes to overall packet rate
    • check against every rule (other packets match after half of the rules)
Performance Evaluation

Changing rules

Rule entry delay without traffic

![Graph showing transaction delay over number of rules and simultaneous calls. The y-axis represents transaction delay in ms, ranging from 0 to 45, and the x-axis represents the number of rules and simultaneous calls, ranging from 0 to 6000.]
Performance Evaluation

Changing rules

Rule deletion delay without traffic

![Graph showing the relationship between the number of rules and transaction delay with simultaneous calls.](image-url)
Performance Evaluation

Throughput

Delay and loss over rule size and rate

→ performance sufficient for 500 simultaneous calls
Performance Evaluation

Throughput

Throughput while discarding bad traffic

rate of illegitimate packets (DoS) increased until 0.1 % loss occured

![Graph showing throughput and transaction delay](image-url)
Performance Evaluation

Measurement summary

Rule management

• effort mainly depends on ruleset size
  reason: translation between kernel and user space representations
• spikes in rule entry delay due to caching effects?
• saltus at ~4096 rules due to paging effects?

Throughput

• sufficient for ~500 calls (pure good traffic)
• for dimensioning: consider max packet rate of bad traffic!
• delay negligible, if not in overload - there are only very small Queues
  → still decent performance for standard hardware
    e.g. enterprise with 20 Mbit/s link: 250 simultaneous calls (each 80 Kbit/s)
    • performance sufficient, even with DoS traffic
    • corresponds to 5000 users (0.05 Erlang)
Possible Improvements

Changes in backend to improve performance

- same API but better mapping to netfilter
- keep it simple: no additional protocol checks in Connttrack (like checking RTP)
Conclusion and Outlook

Conclusion

• API for dynamic firewall control (phapi) designed and implemented
• can integrate with our SIMCO-Server (sourceforge.net/projects/simco-firewall/)
• pinhole api implementation (phapi): www.ikr.uni-stuttgart.de/Content/firewall/
• filter/chains based on linear search perform quite well
• interaction with Conntrack cannot be easily solved (conntrack must be disabled)

Outlook

• interface between Conntrack and backend
  – keep information about mapping between conntrack entry and pinhole
  – stateful fast filtering
  – resolves interaction issue
  – still use iptables chains for large ranges/wildcards
  – optimal mapping? what is large? How costly are filter rules compared to Conntrack entries?
• implementation for other packet filters (OpenBSD, Network Processors, FPGA, ...)

© 2008 Universität Stuttgart • IKR