A userspace API for netfilter control

Netfilter Workshop 2007, Karlsruhe

Sebastian Kiesel, Jochen Kögel, Sebastian Meier, Christian Blankenhorn
Institute of Communication Networks and Computer Engineering
University of Stuttgart
{kiesel, koegel, smeier, blankenhorn}@ikr.uni-stuttgart.de

September 11, 2007
• Problem statement
  - limitations of connection tracking
  - alternatives
• Firewall Control Frameworks: Overview
• Requirements on a Pinhole API
• Pinhole API for netfilter
  - Design considerations
  - Implementation status
  - Mapping of pinholes to netfilter
• Conclusion and Outlook
Scenario

- control flow and RELATED media flow
  - VoIP: SIP and RTP
- strict fine-grained policies
  - not -A OUTPUT -p UDP -j ALLOW
  - more than allow/not allow connection from/to
- more than one border element (load-balancing, protection, multi-homing,..)
Problem Statement

Approach 1: Connection Tracking only

problems
- extensibility/maintainability: new kernel modules for new or changed control protocols
- robustness/security risk: parsing of complex protocols in the kernel
- no authorization/fine-grained policies requires additional internal SIP-proxy/B2BUA
Problem Statement

Approach 2: Application Layer Gateways (ALG)

• SIP-ALG: Session Border Controller (SBC)
  - Processing of signaling and media (all in user space)
  - All RTP routed through ALG independent of IP-Routing
  - SBC needs full application knowledge (RTP codecs, ...)
  - Packet filter in front of SBC: completely open to UDP? Conntrack?
Approach 3: Firewall Control Protocol

- **firewall control daemons**
  - running on firewall machines
  - 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
Approach 3: Firewall Control Protocol - prohibiting flows (IDS)

Firewall control daemon: how to control packet filter?
- calling command line tools
- using libraries (libiptc, nfnetlink)
  ➔ lots of dependencies on filter implementation, libraries, formats, OS
  ➔ general API makes sense
  ➔ detailed requirements? first have a look at firewall control...
Firewall Control Frameworks

Firewall/NAT Control protocol zoo
- IETF MIDCOM Framework
  - Implementation: Simco
- IETF NSIS
  - path-coupled signaling framework (QoS requests, NAT, firewall)
- H.248 MEGACO
  - ETSI: Profile for controlling media relays (BGF)
  - H.248.37: signal SBC to replace addresses for NAT traversal

➤ Focus on firewall control: MSimco, NSIS
MIDCOM Framework (RFC 3303)
- abstract protocol semantics for NAT/FW control
- abstract protocol entities
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)
- Pinhole
  - two "address tuples" (transport protocol, address, prefix, port, portrange)
  - ports and address wildcarding
  - inbound/outbound/bidirectional
  - can be mapped on 5-Tuple with ranges

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
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-15.txt)
  - Authorization based on tokens (draft-manner-nslp-auth-03.txt)
NSIS Firewall Signaling:

- **Pinhole description based on existing flow**
  - `sub_ports`: how many contiguous ports (0..1)
  - Allow/Deny
  - blocking traffic with EXT messages (for whole prefix, port wildcard)
Requirements on an API

There are several reasons for changing packet filter rules dynamically
- firewall control protocols (our motivation)
- ALG implementations
- Intrusion detection systems

Often realized by calling iptables, but libraries available are very specific (libiptc). Strong dependency on filter realization.

➤ Why not designing a common (high-level, userspace) API?
➤ We started based on requirements from a SIMCO-Prototype
(Our) Requirements

- open/close pinholes
- pinhole: 5-Tuple (incl. subnets + port ranges)
  - bidirectional: two pinholes
- independent of filter-implementation (and OS)
- transaction semantics
- no control of whole packet filter, only dedicated rule sets (e.g. one chain)
- fast
  - frequent rule changes (VoIP)
  - high packet rate
Pinhole API for netfilter

Vision

- SIMCO
- NSIS
- NATFW
- IDS

- enterPH
- removePH

uniform interface (OS independent)

Application/Daemon

unpriv. user

priv. user (root)

kernel

- chains
- set+chains
- BSD

- translation plugin according to OS/config/kernel capab.

Have a look into the details..
Interface

Example: C++ interface

```cpp
ruleManager.request(MODIFY_RULESET);
int ruleID1 = ruleManager.addRule(
    "1.2.3.4", 24,
    100, 200,
    "2.3.4.5", 24,
    300, 400,
    IPPROTO_UDP, AF_INET);
ruleManager->commit();

ruleManager.request(MODIFY_RULESET);
ruleManager.delRule(ruleID1);
int ruleID2 = ruleManager.addRule(
    "5.6.7.8", 24,
    100, 200,
    "6.7.8.9", 24,
    300, 400,
    IPPROTO_UDP, AF_INET);
ruleManager->commit();
```
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
  - in intervals (with backoff-Alg)
  - currently: complete rule set
  - libiptc backend performance: changing or rewriting rules takes almost the same time
  - socket communication: reuse of SIMCO message definition + added new control messages
Pinhole API for netfilter

**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
Institute of Communication Networks and Computer Engineering University of Stuttgart

Pinhole API for netfilter

**Backend**

- works on predefined chain
- integrate this chain into your packet filter configuration
- configure the rest of packet filter as you like

Example configuration of a packet filter using phapi

```bash
iptables -N phapi #chain to be used by daemon
iptables -A FORWARD -j phapi
iptables -A FORWARD -j DROP

#starting daemon
#syntax: phapi_backend -s <socket> -u <socket_user> -c <chain_name> [-t <target>]
phapi_backend -s /tmp/phapi -u koegel -c phapi
```
Performance

Measurements with libiptc backend (VoIP Scenario)

- 20 ms packetizing time: 100 pps/call (bidir.), no bursts
- 8 pinholes per call: (asymmetric RTP + RTCP) x 2
- "bad/unwanted traffic" - will be filtered, but
  • also contributes to overall packet rate
  • check against every rule (other packets match after half of the rules)
- entering changed pinhole set into netfilter chain
  • in fixed time intervals (every ??100 ms)
  • effort depends on amount of pinholes
  • effort independent of number of changes
  • at high packet rates
- P4 2,53 GHz
Pinhole API for Netfilter

VoIP-Traffic only
 simultanous calls

Rule-Entry delay
Packet delay
Packet loss

Delay (in ms)
Packet loss (in %)
Number of rules
Summary

Current stage
- preview version: http://www.ikr.uni-stuttgart.de/Content/firewall/
- C++ API
- backend based on libiptc

Open issues
- C interface
- rule optimizer?
- handling TCP direction
- improving performance
- Nat support

How to map pinholes to netfilter?
TCP direction

- meaning of direction different than for UDP: not direction of packet flow but direction of connection establishment
- One TCP-pinhole signaled
  - source->destination (for every packet)
  - destination -> source (RELATED, !--syn)
  - if using conntrack – closing pinhole means removing conntrack entry
    - makes API implementation complicated and dependant on static configuration
  - therefore: first go for simple !--syn
- Two pinholes signaled (bidirectional)
  - two rules, direction of connection establishment does not matter
  - more intelligence in the backend
Improving performance

Criteria
- faster rule changes
- faster filtering

Hash-based?
- exact flow match only (no ranges)
- thus: combination of hash and list
  - pinhole without range: use hash
  - pinhole with range: use list
  - pinhole with small range: several hash-entries (what is small? 4, 10, 100?)
- conntrack? ipset?
1. Conntrack
   - pinholes in conntrack table (permanent/timeout?)
   - already present in most configuration, implicit semantics
     ... --state ESTABLISH, RELATED -j ACCEPT

2. IPset
   - currently no 5-tuple match, extension possible
   - simple configuration, just like using chains
   - fast: 10,000 entries are no problem

**Idea for fast netfilter backend**
   - extension of backend - ipset for small pinholes, chains for ranges
   - how to combine this with TCP-direction-problem?
     • two 5-tuple ipsets + list
     • first set for all pinholes, target: tcp !--syn
     • rule filtering on TCP --syn
     • second set with pinholes allowing SYN
     • ...or extending the 5-tuple ipset with flag for --syn?
Conclusion

- simple Pinhole API
  - 5-tuple + ranges + direction sufficient for most firewall control tasks
  - transaction semantics: defined state and less communication effort
- current prototype implementation phapi
  - daemon + socket communication: privilege separation
  - uses netfilter chains: decent performance, could be better

Discussion

- Additional requirements?
  - rate limiting
  - NAT support
- Better performance by suitable mapping of pinholes to netfilter
  - ipset for 5-tuples? conntrack?
  - suitable for large scale setups?