ALPHA

Johannes Gilger, Florian Weingarten

January 9th, 2009

Adaptive and Lightweight Protocol
for Hop-By-Hop Authentication

Summary

What we did since last meeting

- Solve routing problem :-)
- Implement three way dummy handshake (similar to screamer)
- Implement UDP encapsulation
- Handle multiple endpoints/clients simultaneously
  (with just one Alpha process and without using ugly multi-threading)
- Implement on-the-fly adding/deleting of endpoints

Problems

- kernel seems to handle incoming `tun`-packets incorrect
- TUN/TAP device driver documentation is **really** bad
  (i.e. does not exist!)

Summary

What we did since last meeting

- Solve routing problem :-)
- Implement three way dummy handshake (similar to screamer)
- Implement UDP encapsulation
- Handle multiple endpoints/clients simultaneously
  (with just one Alpha process and without using ugly multi-threading)
- Implement on-the-fly adding/deleting of endpoints

Problems

- kernel seems to handle incoming `tun`-packets incorrect
- TUN/TAP device driver documentation is **really** bad
  (i.e. does not exist!)

Summary

What we did since last meeting

- Solve routing problem :-)
- Implement three way dummy handshake (similar to screamer)
- Implement UDP encapsulation
- Handle multiple endpoints/clients simultaneously
  (with just one Alpha process and without using ugly multi-threading)
- Implement on-the-fly adding/deleting of endpoints

Problems

- kernel seems to handle incoming `tun`-packets incorrect
- TUN/TAP device driver documentation is **really** bad
  (i.e. does not exist!)

Routing problem: Solved

Problem

- Route traffic through tun0 interface, **except** traffic to alpha daemon
- Solution: policy based routing, iptables + iproute2

How?

Mark packets with iptables mangle:

- `iptables -t mangle -A OUTPUT -p udp -m udp -d $DEST ! --dport $PORT -j MARK --set-mark $MARK`
- `iptables -t mangle -A OUTPUT -p {tcp,icmp} -d $DEST -j MARK --set-mark $MARK`

Route marked packets through tun0:

- `ip rule add fwmark $MARK lookup $TABLE`
- `ip route add default dev tun0 table $TABLE`

Routing problem: Solved

Problem

- Route traffic through `tun0` interface, **except** traffic to alpha daemon
- Solution: policy based routing, `iptables` + `iproute2`

How?

Mark packets with iptables mangle:

- `iptables -t mangle -A OUTPUT -p udp -m udp -d $DEST ! --dport $PORT -j MARK --set-mark $MARK`
- `iptables -t mangle -A OUTPUT -p {tcp,icmp} -d $DEST -j MARK --set-mark $MARK`

Route marked packets through `tun0`:

- `ip rule add fwmark $MARK lookup $TABLE`
- `ip route add default dev tun0 table $TABLE`

Routing problem: Solved

Problem

- Route traffic through `tun0` interface, **except** traffic to alpha daemon
- Solution: policy based routing, `iptables` + `iproute2`

How?

Mark packets with iptables mangle:

- `iptables -t mangle -A OUTPUT -p udp -m udp -d $DEST ! --dport $PORT -j MARK --set-mark $MARK`
- `iptables -t mangle -A OUTPUT -p {tcp,icmp} -d $DEST -j MARK --set-mark $MARK`

Route marked packets through `tun0`:

- `ip rule add fwmark $MARK lookup $TABLE`
- `ip route add default dev tun0 table $TABLE`

Routing problem: Solved

Problem

- Route traffic through `tun0` interface, **except** traffic to alpha daemon
- Solution: policy based routing, `iptables` + `iproute2`

How?

Mark packets with iptables mangle:

- `iptables -t mangle -A OUTPUT -p udp -m udp -d $DEST ! --dport $PORT -j MARK --set-mark $MARK`
- `iptables -t mangle -A OUTPUT -p {tcp,icmp} -d $DEST -j MARK --set-mark $MARK`

Route marked packets through `tun0`:

- `ip rule add fwmark $MARK lookup $TABLE`
- `ip route add default dev tun0 table $TABLE`

Routing problem: Solved

Discussion

- best (i.e. only) working way we found
- very flexible, thanks to `iptables`
  (maybe you only want http traffic tunneled? no problem)
- not portable (obviously) :-(
- maybe a bit overkill to use netfilter here
- but implemented modular

Routing problem: Solved

Discussion

- best (i.e. only) working way we found
- very flexible, thanks to `iptables`
  (maybe you only want http traffic tunneled? no problem)
- not portable (obviously) :-(
- maybe a bit overkill to use netfilter here
- but implemented modular

How it looks so far...

Alpha daemon

```
Successfully opened tun device (tun0)
Binding listening socket to port 1234
```

Adding an endpoint

```
$ ./route.sh tun0 vm2 1234
setting routes and iptables rules for 192.168.10.61:1234 (tun0)

$ ./add_host.pl vm2
Adding host ‘vm2’
Sending SIGHUP to alpha
```

Alpha daemon

```
Received SIGHUP!
Adding host vm1 (192.168.10.60). Creating client socket.
```

How it looks so far...

Alpha daemon

```
Successfully opened tun device (tun0)
Binding listening socket to port 1234
```

Adding an endpoint

```
$ ./route.sh tun0 vm2 1234
setting routes and iptables rules for 192.168.10.61:1234 (tun0)

$ ./add_host.pl vm2
Adding host 'vm2'
Sending SIGHUP to alpha
```

Alpha daemon

```
Received SIGHUP!
Adding host vm1 (192.168.10.60). Creating client socket.
```

How it looks so far...

Alpha daemon

```
Successfully opened tun device (tun0)
Binding listening socket to port 1234
```

Adding an endpoint

```
$ ./route.sh tun0 vm2 1234
setting routes and iptables rules for 192.168.10.61:1234 (tun0)

$ ./add_host.pl vm2
Adding host 'vm2'
Sending SIGHUP to alpha
```

Alpha daemon

```
Received SIGHUP!
Adding host vm1 (192.168.10.60). Creating client socket.
```

How it looks so far...

Alpha daemon

```
Successfully opened tun device (tun0)
Binding listening socket to port 1234
```

Adding an endpoint

```
$ ./route.sh tun0 vm2 1234
setting routes and iptables rules for 192.168.10.61:1234 (tun0)

$ ./add_host.pl vm2
Adding host ‘vm2’
Sending SIGHUP to alpha
```

Alpha daemon

```
Received SIGHUP!
Adding host vm1 (192.168.10.60). Creating client socket.
```

How it looks so far... (2)

vm1: Send a packet (which gets routed through `tun0`)

```
$ echo "this is a test" | nc vm2 2323 —u
```

vm1: Alpha daemon

```
Initiating handshake with 192.168.10.61, sending SYN.
Got ACK packet. Handshake with 192.168.10.61 is done! Sending ACKACK.
——tun0——> udp packet from 192.168.10.60 to 192.168.10.61 (ttl 64, 43 bytes)
Sending encapsulated packet to 192.168.10.61:1234
```

vm2: Alpha daemon

```
Got SYN packet. 192.168.10.60 has initiated a handshake! Sending ACK.
Got ACKACK packet. Handshake with 192.168.10.60 is done!
Received encapsulated packet (44 bytes) from 192.168.10.60:46466
<——tun0—— udp packet from 192.168.10.60 to 192.168.10.61 (ttl 64, 43 bytes)
```

vm2: ‘`ngrep -d tun0`’

```
U 192.168.10.60:36788 —> 192.168.10.61:2323   this is a test
```

How it looks so far... (2)

vm1: Send a packet (which gets routed through tun0)

```
$ echo ”this is a test” | nc vm2 2323 —u
```

vm1: Alpha daemon

```
Initiating handshake with 192.168.10.61, sending SYN.
Got ACK packet. Handshake with 192.168.10.61 is done! Sending ACKACK.
——tun0——> udp packet from 192.168.10.60 to 192.168.10.61 (ttl 64, 43 bytes)
Sending encapsulated packet to 192.168.10.61:1234
```

vm2: Alpha daemon

```
Got SYN packet. 192.168.10.60 has initiated a handshake! Sending ACK.
Got ACKACK packet. Handshake with 192.168.10.60 is done!
Received encapsulated packet (44 bytes) from 192.168.10.60:46466
<——tun0—— udp packet from 192.168.10.60 to 192.168.10.61 (ttl 64, 43 bytes)
```

vm2: ‘ngrep -d tun0’

```
U 192.168.10.60:36788 —> 192.168.10.61:2323   this is a test
```

How it looks so far... (2)

vm1: Send a packet (which gets routed through `tun0`)

```
$ echo "this is a test" | nc vm2 2323 —u
```

vm1: Alpha daemon

```
 Initiating handshake with 192.168.10.61, sending SYN.
 Got ACK packet. Handshake with 192.168.10.61 is done! Sending ACKACK.
——tun0——> udp packet from 192.168.10.60 to 192.168.10.61 (ttl 64, 43 bytes)
 Sending encapsulated packet to 192.168.10.61:1234
```

vm2: Alpha daemon

```
 Got SYN packet. 192.168.10.60 has initiated a handshake! Sending ACK.
 Got ACKACK packet. Handshake with 192.168.10.60 is done!
 Received encapsulated packet (44 bytes) from 192.168.10.60:46466
<——tun0—— udp packet from 192.168.10.60 to 192.168.10.61 (ttl 64, 43 bytes)
```

vm2: ‘`ngrep -d tun0`’

```
U 192.168.10.60:36788 —> 192.168.10.61:2323    this is a test
```

How it looks so far... (2)

vm1: Send a packet (which gets routed through tun0)

```
$ echo ”this is a test” | nc vm2 2323 —u
```

vm1: Alpha daemon

```
 Initiating handshake with 192.168.10.61, sending SYN.
 Got ACK packet. Handshake with 192.168.10.61 is done! Sending ACKACK.
——tun0——> udp packet from 192.168.10.60 to 192.168.10.61 (ttl 64, 43 bytes)
 Sending encapsulated packet to 192.168.10.61:1234
```

vm2: Alpha daemon

```
 Got SYN packet. 192.168.10.60 has initiated a handshake! Sending ACK.
 Got ACKACK packet. Handshake with 192.168.10.60 is done!
 Received encapsulated packet (44 bytes) from 192.168.10.60:46466
<——tun0—— udp packet from 192.168.10.60 to 192.168.10.61 (ttl 64, 43 bytes)
```

vm2: ‘ngrep -d tun0’

```
U 192.168.10.60:36788 —> 192.168.10.61:2323    this is a test
```

How it looks so far... (2)

vm1: Send a packet (which gets routed through `tun0`)

```
$ echo ”this is a test” | nc vm2 2323 —u
```

vm1: Alpha daemon

```
 Initiating handshake with 192.168.10.61, sending SYN.
 Got ACK packet. Handshake with 192.168.10.61 is done! Sending ACKACK.
—tun0—> udp packet from 192.168.10.60 to 192.168.10.61 (ttl 64, 43 bytes)
 Sending encapsulated packet to 192.168.10.61:1234
```

vm2: Alpha daemon

```
 Got SYN packet. 192.168.10.60 has initiated a handshake! Sending ACK.
 Got ACKACK packet. Handshake with 192.168.10.60 is done!
 Received encapsulated packet (44 bytes) from 192.168.10.60:46466
<—tun0— udp packet from 192.168.10.60 to 192.168.10.61 (ttl 64, 43 bytes)
```

vm2: ‘`ngrep -d tun0`’

```
U 192.168.10.60:36788 —> 192.168.10.61:2323   this is a test
```

Packets disappear, are ignored, whatever, we didn't know

Problem

- packets arrive at `tun0`
  (according to `tcpdump` or `ngrep`, see previous slide)
- packets arrive in the `PREROUTING` chain of netfilter
  (according to `iptables -j LOG`
- packets do **NOT** arrive in the `INPUT` chain of netfilter
- packets do **NOT** arrive in user space or kernel space
- they just disappear and nothing happens

Solution

- Several days of reading, googling, usenetting, ...
- Lots of nerves and coffee
- One single stupid flag was set wrong: `rp_filter`

Packets disappear, are ignored, whatever, we didn't know

Problem

- packets arrive at `tun0`
  (according to `tcpdump` or `ngrep`, see previous slide)
- packets arrive in the `PREROUTING` chain of netfilter
  (according to `iptables -j LOG`
- packets do **NOT** arrive in the `INPUT` chain of netfilter
- packets do **NOT** arrive in user space or kernel space
- they just disappear and nothing happens

Solution

- Several days of reading, googling, usenetting, ...
- Lots of nerves and coffee
- One single stupid flag was set wrong: `rp_filter`

Packets disappear, are ignored, whatever, we didn't know

Problem

- packets arrive at `tun0`
  (according to `tcpdump` or `ngrep`, see previous slide)
- packets arrive in the `PREROUTING` chain of netfilter
  (according to `iptables -j LOG`
- packets do **NOT** arrive in the `INPUT` chain of netfilter
- packets do **NOT** arrive in user space or kernel space
- they just disappear and nothing happens

Solution

- Several days of reading, googling, usenetting, ...
- Lots of nerves and coffee
- One single stupid flag was set wrong: `rp_filter`

Packets disappear, are ignored, whatever, we didn't know

What is `rp_filter`?

- `rp_filter`: Reverse Path Filter
- `/proc/sys/net/ipv4/conf/tun0/rp_filter`
- protection against incoming IP packets with fake sender address
- if packets are coming in, the source address is checked by the kernel against the routing table
- if the outgoing interface, which would be used according to **the main routing table**, for sending packets to that address, does not match the interface the packet came in, it is assumed to be fake and silently dropped by the kernel
- Of course, this screws up everything for us
  (and everybody else who uses asynchronous routing; so thats good to keep in mind :-))

Solution

- Disable `rp_filter`
- (thats fine, we don't need spoof protection on a `tun` device)

Packets disappear, are ignored, whatever, we didn't know

What is rp_filter?

- rp_filter: Reverse Path Filter
- /proc/sys/net/ipv4/conf/tun0/rp_filter
- protection against incoming IP packets with fake sender address
- if packets are coming in, the source address is checked by the kernel against the routing table
- if the outgoing interface, which would be used according to **the main routing table**, for sending packets to that address, does not match the interface the packet came in, it is assumed to be fake and silently dropped by the kernel
- Of course, this screws up everything for us
  (and everybody else who uses asynchronous routing; so thats good to keep in mind :-))

Solution

- Disable rp_filter
- (thats fine, we don't need spoof protection on a tun device)

Packets disappear, are ignored, whatever, we didn't know

What is rp_filter?

- rp_filter: Reverse Path Filter
- /proc/sys/net/ipv4/conf/tun0/rp_filter
- protection against incoming IP packets with fake sender address
- if packets are coming in, the source address is checked by the kernel against the routing table
- if the outgoing interface, which would be used according to **the main routing table**, for sending packets to that address, does not match the interface the packet came in, it is assumed to be fake and silently dropped by the kernel
- Of course, this screws up everything for us
  (and everybody else who uses asynchronous routing; so thats good to keep in mind :-))

Solution

- Disable rp_filter
- (thats fine, we don't need spoof protection on a tun device)

What's next?

What we got right now

- A fully functional ip tunnel, completely written from scratch. Yeah!
- About 1300 lines of source code, mostly C
- ...

Our goals for the next week(s)

- Start implementing the alpha protocol
- Figure out how to set the right MTU for the `tun` device

What's next?

What we got right now

- A fully functional ip tunnel, completely written from scratch. Yeah!
- About 1300 lines of source code, mostly C
- ...

Our goals for the next week(s)

- Start implementing the alpha protocol
- Figure out how to set the right MTU for the `tun` device

What's next?

What we got right now

- A fully functional ip tunnel, completely written from scratch. Yeah!
- About 1300 lines of source code, mostly C
- ...

Our goals for the next week(s)

- Start implementing the alpha protocol
- Figure out how to set the right MTU for the `tun` device