Topics in Network Intrusion Detection

Vern Paxson

AT&T Center for Internet Research at ICSI
International Computer Science Institute
and
Lawrence Berkeley National Laboratory
University of California
Berkeley, CA

vern@{aciri.org, ee.lbl.gov}

November 5, 2001

Overview:

Why network intrusion detection? Why not?

Styles of approaches.

An example of a NIDS: BRO.

The fundamental problem of evasion, possible solutions.

Detecting activity: sniffers, stepping stones, backdoors.

What can you learn watching a network link?

- Far and away, most traffic travels across the Internet unencrypted.
- Communication is layered with higher layers corresponding to greater semantic content.
- The entire communication between two hosts can be reassembled: individual *packets* (e.g., TCP/IP headers), application *connections* (TCP byte streams), user *sessions* (Web surfing).
- You can do this in real-time.

Tapping links, con't:

- Appealing because it's *cheap* and gives broad coverage.
- You can have multiple boxes watching the same traffic..
- Generally (not always) undetectable.
- Can also provide insight into a site's general network use.

Problems with passive monitoring:

- Reactive, not proactive.
- Assumes network-oriented (often “external”) threat model.
- For high-speed links, monitor may not be able to keep up. Accordingly, monitors often rely on filtering (kernel/BPF). Very high speed: beyond state-of-the-art.
- Depending on “vantage point”, sometimes you see only one side of a conversation (especially inside backbone).
- Against a skilled opponent, there is a fundamental problem of evasion: confusing / manipulating the monitor.

Styles of intrusion detection — *Signature-based*:

Core idea: look for specific, known attacks.

Example: "`src addr` == `dst addr` &&
`src port` == `dst port` $\Rightarrow$ *Land* attack."

Note: Can be at different semantic layers (e.g., URLs).

Pro: good attack libraries, easy to understand results.

Con: unable to detect new attacks, or even just variants.

Styles of intrusion detection — *Anomaly-detection*:

Core idea: attacks are *peculiar*.

Approach: build/infer a profile of "normal" use, flag deviations.

Example: "user `joe` only logs in from host A, usually at night."

Note: works best for *narrowly-defined* entities.

Pro: potentially detects wide range of attacks, including novel.

Con: potentially misses wide range of attacks, including known.

Con: can potentially be "trained" to accept attacks as normal.

Styles of detection — *Activity*- (or *Specification*-) based:

Core idea: piece traffic into events, look for patterns of activity that deviate from a site's *policy*.

Example: "user `joe` is *only* allowed to log in from host A."

Note: this is the approach used by BRO.

Pro: potentially detects wide range of attacks, including novel.

Pro: framework can accommodate signatures, anomalies.

Con: policies/specifications require significant development & maintenance. Harder to construct attack libraries.

Some general considerations about the problem space:

Security is about *policy*.
The goal is risk management, not bulletproof protection.

All intrusion detection systems suffer from the twin problems of *false positives* and *false negatives*.
These are not minor, but an Achilles heel.

Scaling works against us: as the volume of monitored traffic grows, so does its diversity.

$\Rightarrow$ NIDS research "in the lab" is *far removed* from operational reality.

A look at BRO— design goals & constraints (1995):

High-speed, large volume monitoring (FDDI/GigEther).

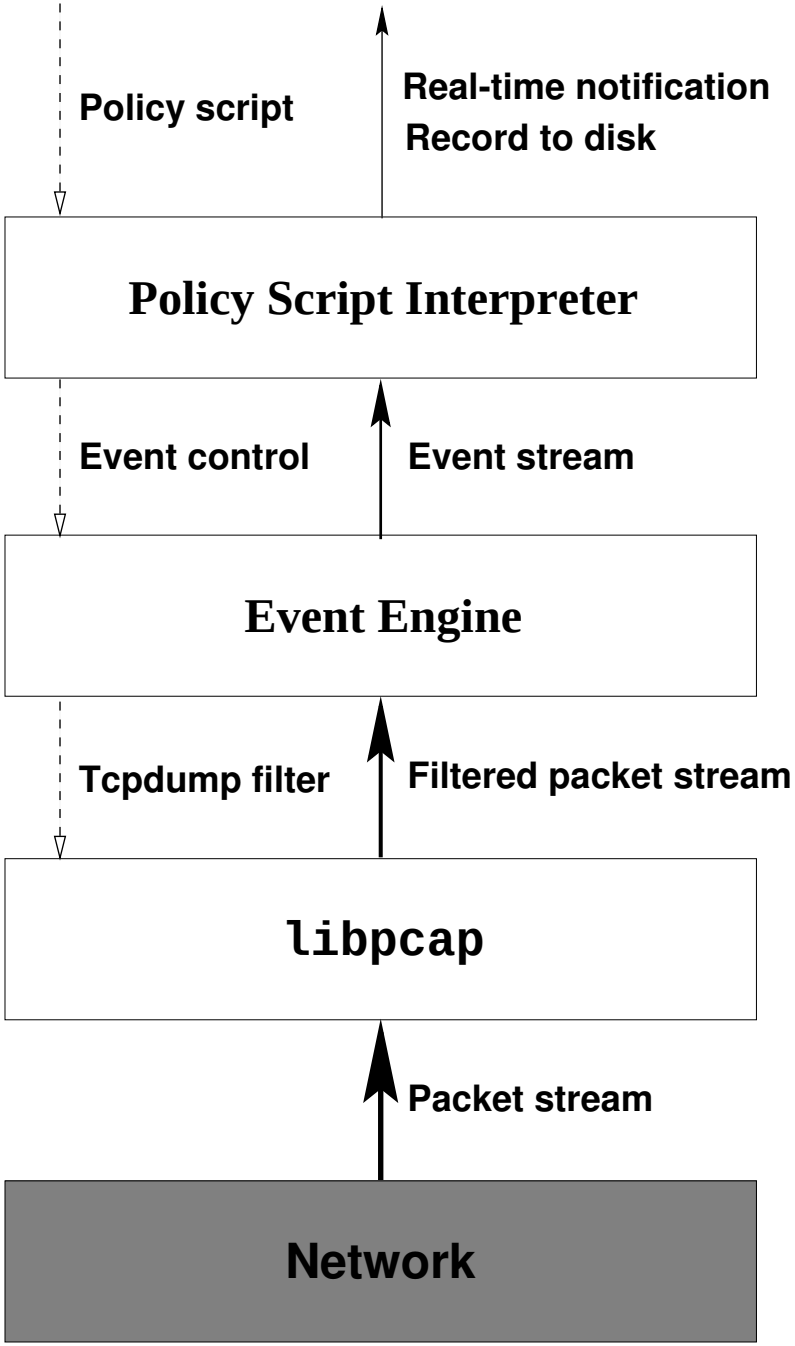
Real-time notification.

Mechanism separate from policy.

Extensible.

Avoid simple mistakes $\Rightarrow$ specialized policy language.

The monitor will be attacked.


Policy script
Real-time notification
Record to disk
Policy Script Interpreter
Event control
Event stream
Event Engine
Tcpdump filter
Filtered packet stream
libpcap
Packet stream
Network

Event engine:

Event engine does generic (non-policy) analysis.

E.g. Connection-level: `connection_attempt`
`connection_finished`

E.g. Application-level: `ftp_request`, `pm_request_getport`,
`login_input_line`

E.g. Activity-level: `login_success`, `stepping_stone`,
`ssh_signature_found`

If you define a handler for a given event, it will be invoked any time the event occurs. Otherwise, event engine skips the work for detecting the event.

The Bro policy language:

Strongly typed $\Rightarrow$ catch errors at compile time.

Arithmetic types, `pattern`, `time`, `interval`, `port`, `addr`.

Records, associative tables & sets:

```
global ftp_sessions:  table[conn_id] of
                                    ftp_session_info
```

Strings are counted rather than NUL-terminated:

```
USER nice\0USER root
```

Analyzers:

For all TCP connections (via SYN/FIN/RST packets):

- start time, duration, service, addresses, sizes
- port, address scanning, including stealth scans

Applications: DNS, FTP, HTTP, NTP, Finger, Portmapper, Ident.

Telnet and RLogin:

`login_successful`, `login_failure`

`activating_encryption`, `login_confused`

$\Rightarrow$ `login_input_line`, `login_output_line`

in first five months of operation,

120 UCB break-ins (60 root compromises)

Teeth:

"`rst`" terminates the local end of a TCP connection via RST packet(s). (Tricky for picky TCP stacks that insist on exact sequence numbers.)

"`drop-connectivity`" talks to border router, throws away given remote traffic: a *reactive firewall*.

Both invoked via `system()`, per arbitrary policy.

At LBNL, 40–70 scans dropped each day.
Routers run with 500–2,000 ACL entries.
When scan blocking fails, mean time to break-in: 4 hours.

Attacks on the monitor:

Overload: make it drop packets, sneak in.

Defense: leave doubt as to monitor's capabilities; shed load.

Crash: make it fault, or run out of resources.

Defense: watchdog timer & backup tracing;

resource management hooks.

Discredit: make it generate zillions of false alarms.

Defense: supple alarm filtering, rule editing.

Subterfuge: fool the monitor.

Example: how to pull out strings from TCP session?

The problem of evasion:

How to detect particular text (e.g., "`USER root`") in a packet?

Easiest: scan for the text in each packet.

No good: text might span multiple packets.

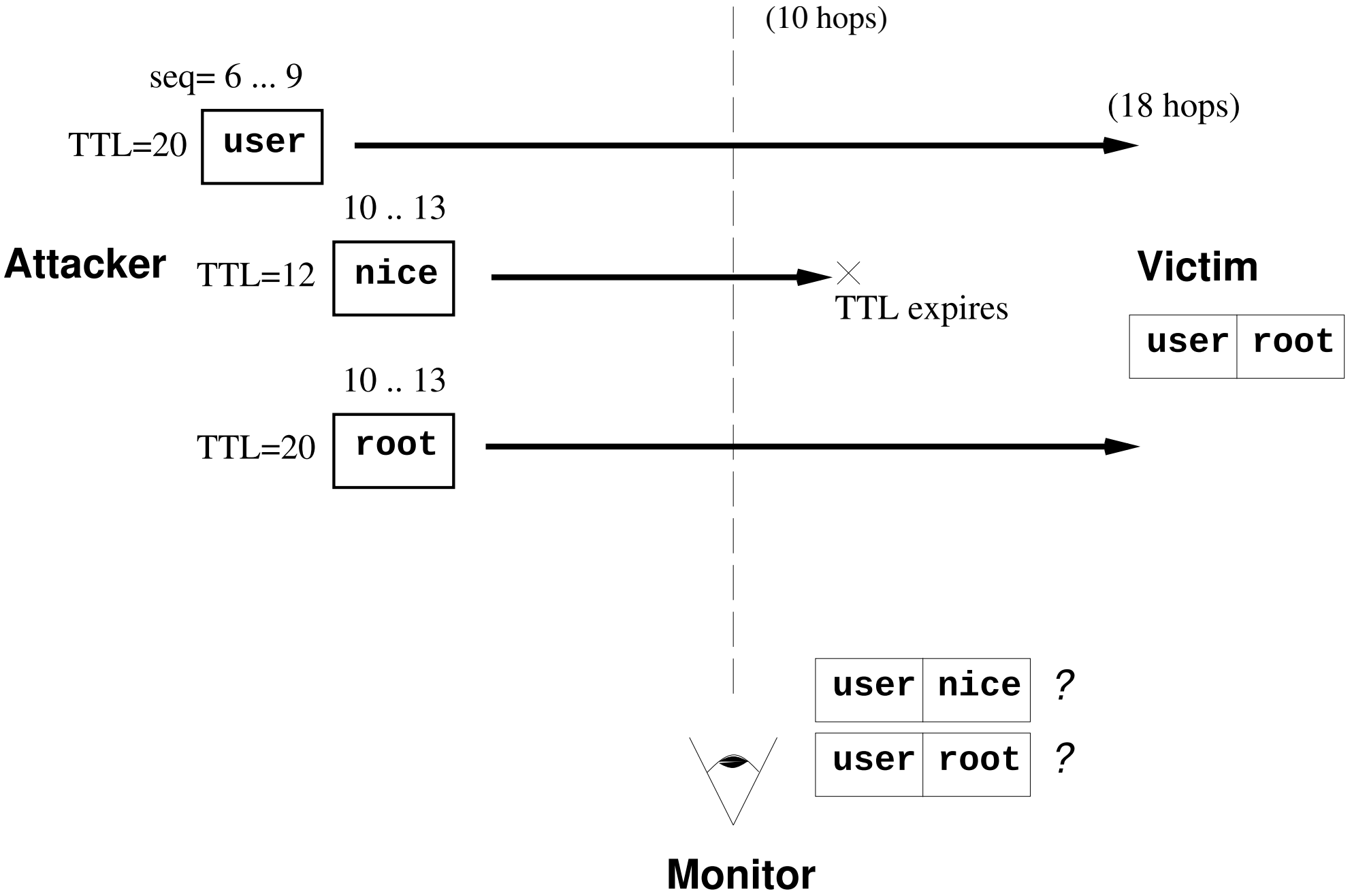
(Major inadequacy of *stateless* NIDS's.)

Okay, then: remember text scanned in last packet.

No good: out-of-order delivery.

Okay, then: fully reassemble the byte stream.

Tricky. Costs state . . . and *still* evadeable.


(10 hops)
seq= 6 ... 9
(18 hops)
TTL=20
user
10 .. 13
Attacker
TTL=12
nice
TTL expires
Victim
user
root
10 .. 13
TTL=20
root
user
nice
?
user
root
?
Monitor

Crud seen on a DMZ:

Storms of 10,000+ FIN or RST packets, due to TCP bugs.

Storms due to foggy days.

Private addresses leaking out.

Legitimate tiny fragments.

Fragments with DF set.

Overlapping fragments.

TCPs that acknowledge data that was never sent (!).

TCPs that *retransmit different data than sent in the first place*.

The problem of evasion, con't:

- Many evasions look anomalous, but when monitoring a high volume traffic stream, you already see *lots of anomalous-but-benign junk.*
- Problem is pervasive. See "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Ptacek/Newsham:
  http://www.aciri.org/vern/Ptacek-Newsham-Evasion-98.ps
- Evasion "toolkits" have been available for a number of years. Not yet widely used.
- As in much of intrusion detection, likely arms race.

Possible ways to resist evasion:

- Exercise great care when developing IDS, attending to all possible exceptional events.
- Deploy sensors on the end hosts.
- Bifurcating analysis. E.g., for "`rob<`*`DEL`*`><`*`BS`*`><`*`BS`*`>ot`", examine directly; as "`ro<`*`BS`*`><`*`BS`*`>ot`"; and as "`root`".
- Understand network and end-host details sufficiently well to resolve ambiguities.
- Deploy a traffic "normalizer" (or "scrubber") to remove ambiguities from traffic stream.

Traffic normalizers — defending against subterfuge:

Idea: “bump in the wire” removes ambiguities from traffic stream.

Examples:

- rectify inconsistent IP fragments
- drop excessively long fragments
- regenerate TTLs (!)
- clear Don’t-Fragment
- shed load under flooding
- defend patsy, victim from stealth port scanning
- make TCP RSTs reliable
- 60+ more

Detecting activity — sniffer detection:

Depending on your threat model, you can often get a lot of mileage out of detecting *evidence of a compromise* rather than the attack itself.

E.g., at LBNL, inbound IRC = break-in.

Another form: sniffer detection.

- e.g., via *increased ping times*
- e.g., via *observing reverse DNS queries*
- e.g., via *transmitting bogus username/password pairs*
- note: works for bad guys detecting IDS, too.

Detecting "stepping stones":

Internet attacks invariably do not come from the attacker's own personal machine, but from a *stepping-stone*: an intermediary previously compromised.

Furthermore, usually it is a *chain* of stepping stones.
Manually tracing attacker back across the chain is virtually impossible.

So: want to detect that a connection going into a site is closely related to one going out of the site.

Detecting stepping stones, con't:

Approach #1 (Staniford/Heberlein, 1994):

Look for similar text content.

For each connection, generate a thumbprint (24 bytes) summarizing per-minute character frequencies.

Connections with similar thumbprints become likely stepping stone candidates.

Detecting stepping stones, con't:

Approach #2 (USAF, 1994):

- Break-in to upstream attack site.
- Recurse.

Detecting stepping stones, con't:

Approach #3 (Zhang/Paxson, 2000):

Leverage unique on/off pattern of user login sessions.

Look for connections that end idle periods at the same time.

Two idle periods correlated if ending time differ by $\leq \sigma$ sec.

If enough idle periods coincide $\Rightarrow$ stepping stone pair.

For $A \rightarrow B \rightarrow C$ stepping stone, just 2 correlations suffices.

(For $A \rightarrow B \rightarrow \ldots \rightarrow C \rightarrow D$, 4 suffices.)

Detecting stepping stones, con't:

$\Rightarrow$ Works very well, *even for encrypted traffic*.

But: easy to evade, if attacker cognizant of algorithm.

And: also turns out there are frequent *legit* stepping stones.

Detecting backdoors:

“Backdoor”: a service installed on a compromised machine to allow the attacker to surreptitously return.

How to find access to these against sea of background traffic?

General algorithm for interactive traffic (Zhang/Paxson 2000):

- look for frequent small packets
- look for small packets with large interarrivals

Detecting backdoors, con't:

Protocol-specific: SSH, Rlogin, Telnet, FTP.

Algorithms also amenable to filtering for large perf. gain:

e.g., `tcp[(tcp[12]>>2):4] = 0x5353482D and`

`(tcp[((tcp[12]>>2)+4):2] = 0x312E or`

`tcp[((tcp[12]>>2)+4):2] = 0x322E)`

Detecting backdoors, con't:

Plus: a hack for detecting some *root* backdoors ("# ").

$\Rightarrow$ Found 437 root backdoors in single 24-hour period at UCB.

Also recognizers for non-interactive protocols:

SMTP, Napster, Gnutella.

In general, algorithms perform quite well.

And: can employ filtering with little loss of accuracy.

But: find many *legit* backdoors.

Summary points:

- The Internet is a frothing sea of attacks.
- You can detect a whole lot by piecing together judiciously filtered network traffic into events reflecting activity . . .
- . . . but there are significant problems with evasion leading to an arms race.
- Traffic contains much more diversity/junk than you'd think.
- The endpoint host is a great location to look for attacks.
- Increasingly, NIDS need to be supplemented by an active forwarding element.