## TCP/IP Overview and Vulnerabilities

We shall examine, in some detail, aspects of the widely deployed TCP/IP internetworking suite that make it vulnerable to attack.

While the TCP/IP suite works extremely well in practice, it is the 'trusting' nature observed in the suite's history and evolution that has recently exposed it to attackers.

We need to examine each of the four layers of the TCP/IP suite to locate its potential vulnerabilities:

- *application layer* protocols, such as telnet, FTP, HTTP, and SMTP, run on (possibly remote) machines to which attackers may not otherwise have physical access. On a case-by-case basis, each of the application services may need to authenticate its remote client, and may use local operating system authentication to perform this, or (dangerously) employ its own mechanism.

  Individual applications offering the networked services are themselves also vulnerable - they may have been poorly written (coded), exposing them to attacks which makes them perform in a manner outside of their expected domain.

- *transport layer* protocols, primarily provided by the reliable, streaming*transport control protocol* (TCP), and the *user datagram protocol* (UDP) meet the data delivery requirements of most Internet applications.

  However, their very design introduces vulnerabilities, because applications and operating systems *expect* the protocols to perform in certain ways. Incorrect interpretation (coding) of protocol RFCs, or attacks against well known sequences of actions in protocols, makes them perform not as expected, or not at all.

## TCP/IP Overview and Vulnerabilities, *continued*

Examining each of the four layers.....

- the *Internet layer* protocols primarily consist of the *Internet protocol* (IP), and the *Internet Control Message Protocol* (ICMP) provide the actual routed delivery of messages between source and destination, and provide only a basic network management function by reporting any observed errors.

  IPv4, particularly, is vulnerable to attack and may be exploited to *not* deliver messages, deliver messages to the wrong destination, or 'confuse' a destination to the extent that it may stop providing any service.

  As examples, IP datagrams may be transmitted from one (attacking) host while claiming to be from another, and forged ICMP messages may make a destination network or host appear unreachable.

- *physical layer* protocols are not strictly part of the TCP/IP suite, but define how packets or frames are received via hardware, and provided to the IP (software) layer above. By its nature, interface hardware must see *all* packets destined for, or *passing by*, the interface, and most hardware may be configured by software (the operating system) to report all activity seen.

  Trivially, on a *shared* network, an operating system (and probably some of its programs) may capture all packets that are visible on a network.

In combination, we have multiple points of vulnerability *in the network protocols themselves*. This is before we consider that the *network* makes hosts more vulnerable to remote attack.

In addition, each operating system's implementation of the TCP/IP stack has its own idiosyncrasies. Specifically, each operating system responds differently to a variety of *malformed* packets. Software performing *protocol fingerprinting* determines an operating system from the way it 'appears' externally.

## Packet Sniffing

Most computer networks consist of many personal computers or workstations connected via a *shared* local area network (LAN and WLAN) segments. Sharing, of course, means that computers can receive information that was intended for other machines.

To capture the information traversing the network is termed *sniffing*.

The most popular form of local LAN topology, Ethernet, works by transmitting addressed packets via a shared cable. The Ethernet network interface card (NIC) in the intended destination computer sees all packets, but on seeing one with the NIC's unique 48-bit address, the NIC will copy the entire packet to the operating system software for analysis and eventual delivery to application programs.

There are two main problems with Ethernet's approach:

- most Ethernet NICs can be placed in *promiscuous mode*, which results in all observed packets being sent to the operating system,

```
root>  ifconfig eth0 promisc
root>  ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:90:27:62:58:84
          inet addr:130.95.1.8  Bcast:130.95.1.255  Mask:255.255.0.0
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          .....
```

  Many rootkits will replace the *ifconfig* program (an abbreviation for *interface configuration*) to avoid the simple detection of interfaces in promiscuous mode.

- and, most Ethernet NICs permit their NIC address to be modified, programatically, and so one Ethernet NIC could (accidentally or deliberately) be given the MAC address of another.

## Packet Sniffing, *continued*

A variety of hardware and software tools are termed *packet sniffers*:

- Packet sniffer - originally a trademark of Network Associates, denotes any hardware or software tool that can capture packets from the network, by setting a node's Ethernet card to report all packets to the system/application regardless of the packet's destination MAC address.

- Network analyzers - tools that monitor network traffic and devices with the goal of alerting the network manager of problems (too much traffic, failed responses from known devices, IP address allocation concerns).

- Protocol analyzers - tools that capture network packets, providing some level of formatting for those packets, allowing the user to analyze/visualize packets *post-hoc*.

Typical uses of such programs, both practical and sinister, include:

- Automatic sifting of clear-text passwords and usernames from the network. Used by attackers to break into active accounts, and remote systems,
- Conversion of data to human readable format so that people can read the traffic,
- Fault analysis to discover problems in the network, such as why computer A can't talk to computer B,
- Network intrusion detection in order to discover attackers, and
- Network traffic logging, to create logs that attackers can't break into and erase.

## TCP/IP port scanning

Using *port scanning* an attacker tries to identify, which services are supported from a potential target host. Whenever an *active port* is located, an attacker may attempt to further determine the version number of any active server/service.

Although port-scanning software such as *nmap* may be used, much of the information about active ports can be determined by a simple tool like *telnet* as well.

What information does an attacker learn from port scanning?

- if an attacker learns that a port is *open*, they can actually connect to the detected port.,
- if an attacker learns that a port is *closed*, they learn that no service is listening to that port,
- additionally, some scanning software reports ports as *filtered*, indicating that a connected attempt was terminated with a `RESET` or timed out.

For example, the naive *TCP Connect Scan* completes the TCP three-way-handshake. A `SYN` packet is sent to the system and if a `SYN/ACK` packet is received, it is assumed that the port on the system is active. If a `RST/ACK` packet is received, it is assumed that the port on the system is not active.

Attackers may further attempt to hide their scans by:

- scanning through ports very slowly, and certainly not in numercial order - unless on a very quiet system, these will not be detected.
- perform hundreds of scans simultaneously from hundreds of random/spoofed IP addresses. The target host will know they are being scanned, by not from where.

## Stealth port scanning

*Stealth scanning* involves searching for open ports, but without actually creating a connection.

*Half-open scanning* only performs the first part of TCP/IP handshake. It sends a `SYN` flag and awaits a reply - a reply with the `SYN` flag set reports an open port, with the `RST` flag set reports an inactive port. Half-open scanning is favoured by potential attackers because (by default) nothing is logged.

Stealth scanning determines a port's status by sending different combinations of TCP options. For example, according to RFC-793 a conforming TCP/IP stack should:

- send back a `RST` packet when they receive a `FIN` packet for a specific closed port (the TCP FIN Scan),
- send back a `RST` packet when they receive a `FIN/URG/PUSH` packet (TCP Xmas Scan), and
- send back a `RST` packet for all TCP ports closed when they receive a packet without any IP flags set (TCP Null Scan).

Implementing a *stealth scan detector* requires kernel-level programming. We need to detect obvious signatures such as:

- several packets from the same source address to different destination ports within a short period of time,
- connection attempts that are not completed with a certain timeout, or
- a `SYN` to a non-listening port.

Even with IP spoofing, naive attacks may themselves leak information, such as a correct `TTL` field indicating the distance to the attacker.

## Internet protocol (IP) spoofing

The *spoofing* of IP packets allows an intruder on the Internet to effectively impersonate a local system's IP address.

In general, IP spoofing and related attacks are possible because programs (maybe requiring superuser access) can open *raw sockets*, create, and send malformed IP packets.

An attacker uses source address spoofing for two reasons:

- to gain access to resources that only accept requests from specific source addresses, or
- to hide the source of an attack by directing the blame at others.

Note that some of these attacks employing these mechanisms are possible even when no reply/response packets can be routed back to the attacker.

If other local systems perform simple session authentication based on the IP address of a connection (e.g. an *rlogin* with `.rhosts` or `/etc/hosts.equiv files` under Unix), they will believe incoming connections from the intruder actually originate from a local 'trusted host' and may not request a password.

Other services, such as the Network File System (NFS), Server Message Block (SMB), and TCP wrappers all include the source address (or system name, in the case of NFS) as part of the access control checks.

It is possible for forged packets to penetrate firewalls based on packet-filtering routers if the router is not configured to block incoming packets with source addresses in the local domain.

## UDP Packet Spoofing

The User Datagram Protocol (UDP, RFC-768) is a lightweight transport protocol built on top of IP. UDP achieves extra performance from IP by *not* implementing some of the session-based features a more heavyweight protocol (like TCP) offers, and typically sees twice the throughput. Specifically:

- UDP allows individual packets to be dropped (with no retries),
- packets may be received in a different order than sent, and
- applications using UDP, typically, do not establish a protocol-level session with their peers. Each request and reply pair are often independent.

An attacker may attack a UDP service *because* of these properties - the attacker is unconcerned about reply packets.

For example, the Network File Service (NFS) employs UDP to 'import' and 'export' file systems. NFS requests, to write, delete, or change file attributes are *atomic*, and can fit in a single UDP packet. Replies only return a simple `OK` and status.

A poorly configured system may permit NFS-based files to be visible to external hosts. An attacker may employ IP source spoofing over UDP, to modify or delete a file.

At the same time, an attacker may also spoof their own source addresses in attacks where reply packets are not important. The attacker does not care about the `OK` response!

## TCP/IP Sequence Number Attacks

We'll consider a representative problem with TCP/IP by examining how TCP/IP establishes sessions between endpoints.

A *three-way handshake* is employed in the TCP *open* sequence.

If machine `A` wishes to establish a connection with machine `B`, `A` transmits the following message:

```
A->B : SYN, ISNa
```

This initial packet request has the *synchronize sequence number* bit (`SSN`) set in its header, and an initial 32-bit unsigned sequence number $ISN_a$.

`B` replies with:

```
B->A : SYN, ISNb, ACK(ISNa)
```

to provide its own initial sequence number, $ISN_b$, and to acknowledge $ISN_a$.

`A` will finally acknowledge $ISN_b$ with

```
A->B : ACK(ISNb)
```

and the connection is established.

This session establishment is considered secure, provided that the initial sequence numbers are so random that they cannot be guessed. If strictly conforming to RFC-793, each TCP/IP implementation is expected to employ its sequence number as a 32-bit counter, modified every 4usec.

## TCP/IP Sequence Number Attacks, *continued*

Traditional BSD-derived implementations only change the 2nd byte of the sequence number every second, and each new connection changes it by 64. An attacker, having established a valid connection, is able to 'guess' the next number to be used.

*A series of well known attacks exploit the non-randomness of the initial sequence numbers.*

The attacker, `C`, establishes a valid connection with `B`, thus determining one of `B`'s 'current' values for $ISN_b$. The attacker, `C`, now impersonates `A` by sending a packet to `B`, but by setting `A`'s NIC address in the Ethernet packet:

```
C(as A)->B : SYN, ISNc
```

`B` replies with

```
B->A : SYN, ISNb* , ACK(ISNc)
```

to the true machine A. `C` will probably *not* see this message `B->A`, but can guess the value of $ISN_{b*}$. `C` now sends

```
C(as A)->B : ACK(ISNb*)
```

and `B` believes that it has a valid connection with A. A is confused as to why it received `B->A`, and may choose to ignore it, or inform `B` (with a `RESET` packet) that something is amiss.

If `A` chooses to ignore the packet `B->A`, then `C` can continue to send packets to `B`, assuming `A`'s identity. If `C` *can* see all replies from `B->A` in the session, then `C` can fully masquerade as `A`, while `A` ignores the transmissions of which it is not a part.

## Denial of Service (DoS) Attacks

Denial of service (DoS) attacks using source address spoofing became popular in 1997, using tools to send thousands of packets to a target system.

A *denial of service* attack is characterised by attackers' explicit attempts to prevent or delay legitimate users from using a service.

Examples include :

- attempts to *flood* a network, thereby preventing or delaying legitimate network traffic,
- attempts to disrupt connections between two machines, thereby preventing access to a service,
- attempts to prevent a particular individual from accessing a service, and
- attempts to disrupt service to a specific system or person.

Often, the source address of these packets is spoofed, making it difficult to locate the real source of the attack.

## The *smurf* DDoS Attack

In the *smurf* DDoS attack, the attacker provides a spoofed source address, when sending an ICMP echo, or *ping*, to an IP broadcast address as the destination
*(The name smurf was adopted after the name of the blue cartoon characters who tended to flood into all locations)* :


- the attacker sends *ICMP Echo Request* packets where the source IP address has been forged to be that of the target of the attack.
- the attacker sends these ICMP datagrams to addresses of remote LANs' broadcast addresses, using so-called directed broadcast addresses. These datagrams are thus broadcast on the LANs by the connected router,
- all the hosts which are alive on the LAN each pick up a copy of the *ICMP Echo Request* datagram, and sends an *ICMP Echo Reply* datagram back to what they think is the source.
- the attacker can use large packets (typically to the Ethernet 1500 byte maximum) to increase the effectiveness of the attack.

The use of broadcast addresses for protocol attacks is termed *amplification*.

The *smurf* attack has 3 types of victims:

- the single destination victim of the attack,
- a network abused (temporarily) to amplify the attack, and
- (always) the host harboring the attacker.

One way to defeat smurfing is to disable IP broadcast addressing at each internal network router, however this strictly violates RFC-1812, *'Requirements for IP Version 4 Routers'*.

## The SYN-Flood Attack

We recently saw how the standard TCP/IP session establishment sequence may be used by an attacker to establish one half of a valid connection with a target system.

However, an attacker may choose the `SYN`-Flood, or *half-open*, attack:

- the attacker (client) sends a `SYN` request to the server,
- the server records the request on a queue of connections waiting to complete, replies with a `SYN/ACK` packet, and eagerly awaits the final `ACK` reply.
- however, the attacker does not send the `ACK` reply. Instead, the attacker sends another, actually hundreds of, `SYN` requests with different source forged address.

Fast TCP session establishment is considered vital, but operating systems allocate only a small number of these 'half-open' sockets, before running out of resources. The release of these incomplete 'half-open' sockets is slow (30secs), and so an attacker can quickly exhaust the supply of buffers which are pre-allocated.

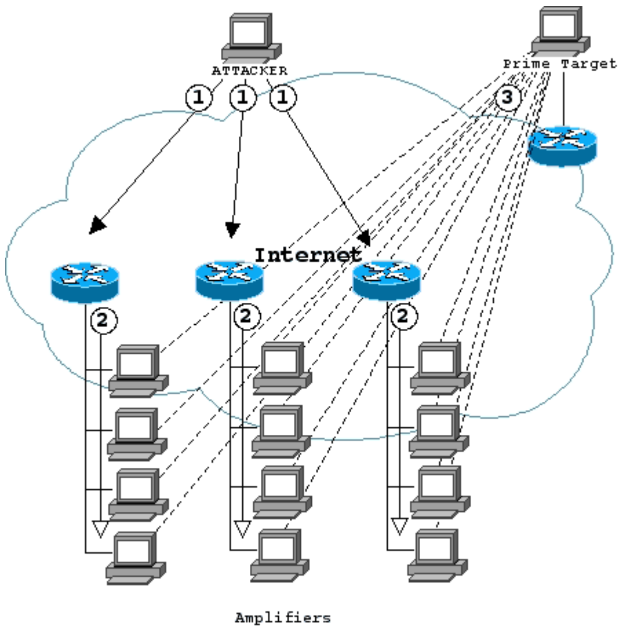
To avoid `SYN`-Flood attacks, modern operating systems will now *not* employ large number of 'half-open' sockets for new connections.

Instead, they will encode and save the opening details (such as the client's IP address) as a 32-bit number, and use this as the initial sequence number in the `SYN/ACK` reply. Only if the final `ACK` reply returns, will socket resources be allocated.

## Distributed Denial of Service (DDoS) Attacks

In a *distributed denial of service* (DDoS) or a *packet storm* attack, an attacker will flood a single system with 'junk' packets to consume bandwidth - preventing legitimate packets getting through.

Using only a single attacker, the effect of the attack is greatly multiplied using attack servers termed *agents*, *zombies*, *daemons* (in the *trinoo* attacks) and *servers* (in the *TFN* attacks).




Attacks are launched simultaneously from hundreds of 'remote-controlled' attack servers. The attacker must first gain access to the hundreds of agent machines, but will use scripts to locate many machines with the same vulnerability.

A single trojan program will typically be installed on each of the agent machines, and *triggered* days or months later by a single UDP or ICMP packet to the agent. All agents will then launch their attacks, using source spoofing, on a single target.

The increased use of ADSL and 'always on' connections, increases the opportunity of DDoS attacks.

RFC-2267 was written in response to this type of attack, suggesting that ISPs should filter traffic and drop any packets with spoofed source addresses. In practical terms, this has proven difficult.

## Security at Network Boundaries

Whereas many forms of network-based attacks can come from *within* our own LANs, the greatest opportunity is provided to an attacker who connects to the LAN from the wider Internet.

Attacks from the Internet can, of course, attempt to bypass the user- or system-level security of a single machine, or possibly undertake a denial-of-service attack on the LAN itself.

In general, we wish to develop security practices at the *boundary* between a LAN and the wider Internet, to constrain the types of network traffic that may cross the boundary.

Specifically, we would like to:

- control network traffic based on both senders' and receivers' network (IP) address,
- control network traffic based on requested services (IP ports),
- not expose our LAN topology to the wider-Internet, hiding hostnames, addresses, and available services,
- constrain some network traffic based on its *content*,
- only permit internal access from remote users and services, based on their verified identities and (possibly) location, and
- log all Internet connections, attempts, and (suspect?) traffic.

We may have political and administrative control of 'both ends' of a permitted connection, but require that connection's traffic to cross the 'unfriendly' Internet.

## Packet filtering at network boundaries

Like most texts, we shall use the term *firewall*[1] to describe any network device, appliance, or specially configured computer which protects the boundary of an internal network.

Specifically, we shall describe firewalls as software devices through which all network packets must pass, both incoming and outgoing.

Providing a single ingress point to an internal network clearly provides a single opportunity to apply a consistent policy to all network traffic.

The practices of:

- *end-runs*, with which a computer can access the Internet without passing its traffic through the firewall (for example, with a modem or wireless connection), and
- *traffic tunneling*, with which users or applications can embed certain types of unwanted network traffic within permitted protocols (for example, uploading a complete file via a web-based CGI program on a host not permitting HTTP's POST command),

often circumvent the purpose or effectiveness of having a firewall.

[1] The origin of the term *firewall* is variously described in texts, including the iron plates separating train-drivers from the firebox, car drivers from the engine, and even the walls of castles, from which arrows were fired through narrow slits.

## Packet filtering at network boundaries, *continued*

Depending on the provided bandwidth to and from an internal network, e.g. from home over a modem and PPP (56Kbps, 100 packets/sec), or an ADSL or (now) NBN router (3-100Mbps, 250,000 packets/sec), a firewall may be:

- part of a traditional, single, workstation (protecting itself),
- a computer or device protecting several other workstations, or
- a dedicated device doing nothing else but protecting other hosts.

Because a traditional computer, acting as a firewall, must inspect each packet entering and leaving the internal network via a number of different network interfaces (for example, modem, wired-Ethernet, wireless-Ethernet), they must implement and respond to security policies as quickly as possible.

Such requirements usually place the responsibilities in the operating system kernel, with user-level programs used to set, modify, and enquire about the current state.

This is in contrast to popular 'personal firewall' software for home computers - generally user-level programs to which an operating system passes packets for inspection.

Such personal firewall programs, often driven by GUI-based software, run more slowly.

## Possible packet filtering criteria

Network packets may be *filtered* on a number of criteria, such as their routing properties (for IP and ICMP), and transport and service properties (for TCP and UDP).

By examining the *headers* of TCP/IP traffic, we can detect obviously falsified traffic:

- filter on each IP packet's *source address*. Packets which arrive on a network interface connected to the *outside* of our internal network (i.e. the Internet) and announce their source address as being from the internal network, probably have spoofed source addresses.
- filter on each IP packet's *destination address*. Packets destined for an internal network address should not leave the network via an external interface.
- filter based on specific low-level routing or transport protocols, such as denying all ICMP or UDP traffic from leaving,
- filter based on application protocols, such as permitting HTTP and FTP requests to leave, but not permitting NFS mount requests to enter, and
- filter based on recent activity. *Stateful filtering* (or *stateful inspection*) has knowledge of recent traffic; for example, stateful FTP filtering permits an *incoming* FTP data-connection request, only if a corresponding *outgoing* control-connection already exists.

## Developing a Firewall Policy

The establishment of a *firewall policy* simplifies the practice of deciding what traffic to permit and what to filter.

Moreover, a consistent, and consistently applied, policy is a strong argument by system administrators to deny individual requests for new small holes in the firewall by individuals.

Surprisingly the 'firewall community' is divided on default behaviours. Either:

- 'that which is not expressly forbidden is permitted', or
- 'that which is not expressly permitted is forbidden'.

There exists a clear balance between security and user freedoms, and for many organizations (e.g. freedom-loving universities) there is often no simple answer.

However, it is unwise (ignorant) to consider that an attack on *external* hosts and networks could not be launched *from* within your internal network.

For this reason, conventional wisdom says we should have mirrored denial policies filtering traffic *leaving* our networks.

## Packet filtering with *iptables*

*iptables* is currently considered the state-of-the-art in programmable firewall software, recently replacing similar, but deficient, software named *ipfw* and *ipchains*. *iptables* is very similar to earlier software, but also provides *stateful* control over network traffic.

*iptables* actually consists of two software components:

- the *iptables* application program, controlling the set of rules and policies to be enforced, and
- *netfilter* software configured as part of an operating system kernel (compiled into the kernel) to control IP traffic on several network interfaces. The *netfilter* modifications have a long history from BSD Unix, and support both IPv4 and IPv6 protocols, including IPsec encrypted protocols.

Some informative block diagrams:

IPtables-1, IPtables-2, and IPtables-3.

In combination, the *iptables* software provides a variety of mechanisms to*filter* packets, perform *network address translation*, and to *mangle* packet headers. Three *rule tables*, named `filter`, `nat`, and `mangle`, are employed to perform these functions.

Each table of rules has a number of built-in *rule chains* (or lists), which provide sequences of rules to be 'evaluated', in order, until it is decided what should happen to an individual packet.

The standard `filter` table provides default chains named `INPUT`, `FORWARD` and `OUTPUT`, and we'll initially focus on these.

## Packet lifetimes using *iptables*

Consider the 'lifetime' of a single packet as it enters and traverses a firewall:

- the packet could have originated on the firewall host (from a locally running program) and be destined for another host. *iptables* filters these packets using its `OUTPUT` chain of rules before they are retransmitted via an outgoing network interface.

- the packet could have originated from outside of the firewall host, and be destined for processes on the firewall host.

  *iptables* filters these packets using its `INPUT` chain of rules as soon as they arrive via one of the firewall's incoming network interfaces, or

- the packet could have originated from outside of the firewall host, and be destined for another host. *iptables* filters these packets using its `FORWARD` chain of rules as soon as the packet arrives via an incoming interface, and before it is retransmitted on an outgoing interface.

Of note, this generic approach permits the *iptables* software to act on a single workstation with a single network interface (such as an ADSL router link) protecting itself, or as a specific firewall device with several (Ethernet) network interfaces protecting a whole internal network.

## An introduction to filtering rules

We will follow the development of filtering rules for a simple (home) computer with a single network interface. Initially, we'll just consider packet filtering.

Firstly, define the *internal* and *external* networking interfaces that we have, *flush* any existing *iptables* rules for the `filter` table, and define the *default policy* for each chain:

```
INT=ppp0
EXT=ppp0

/sbin/iptables -t filter -F
/sbin/iptables -t filter -X

/sbin/iptables -t filter -P INPUT   DROP
/sbin/iptables -t filter -P FORWARD DROP
/sbin/iptables -t filter -P OUTPUT  DROP
```

## An introduction to filtering rules, *continued*

We'll next create a new rule-chain of *named* rules in the `filter` table. These can be considered as similar to a method, or procedure, of new rules to be evaluated under certain conditions.

We then *append* individual new rules to this named rule-chain:

```
/sbin/iptables -t filter -N myrules

/sbin/iptables -t filter -A INPUT -j myrules
/sbin/iptables -t filter -A FORWARD -j myrules
```

In addition, we can decide how to manage individual packets based on the protocols (`TCP`, `UDP`...) being used, or the services (ports) requested:

```
iptables -A INPUT -p tcp --dport 22 -j ACCEPT
```

We also wish to *log* all packets that our firewall drops, but we don't wish an attacker to flood our machine's logfiles:

```
iptables -t filter -A INPUT \
    -m limit --limit 15/minute \
    -j LOG --log-prefix "suspicious, dropped"
```

These details are logged via the standard `syslogd` mechanism.

## Examining packets on specific interfaces

For brevity, we'll now omit the use of the `-t filter` options, as the filtering table is the obvious default. In each of these examples, we *append* some specific rules to our named chain `myrules`:

- accept existing, established, connections arriving over the external interface:

```
iptables -A myrules -i $EXT -m state \
             --state ESTABLISHED,RELATED -j ACCEPT
```

- permit (we say `ACCEPT`) new packet sequences to leave our machine if they have *not* come from the external interface (i.e. they are from the internal interface, or from local processes):

```
iptables -A myrules -i ! $EXT -m state --state NEW -j ACCEPT
```

- do not permit (we say `DROP`) new packets, or ones with invalid option bits in their headers (such as the XMAS port-scan), that arrive via the external interface. In addition, we log the details before the packet is dropped:

```
iptables -A myrules -i $EXT -m state \
    --state NEW,INVALID -j LOG --log-prefix "dropped"

iptables -A myrules -i $EXT -m state \
    --state NEW,INVALID -j DROP
```

## IP Masquerading

IP masquerading or *network address translation* (NAT) is a technique employed within a firewall, or border gateway, to translate, or map, one set of IP addresses (usually private) to another (usually public).

To use NAT, the firewall connecting the *internal* LAN to the *external* Internet will have (at least) two network cards, each with their own IP address:

- on the Internet side, the machine will use a fully-routable address assigned by an ISP.
- on the LAN side, it will have an address from the *non-routable* addresses, defined in RFC 1918 *'Address Allocation for Private Internets'*:

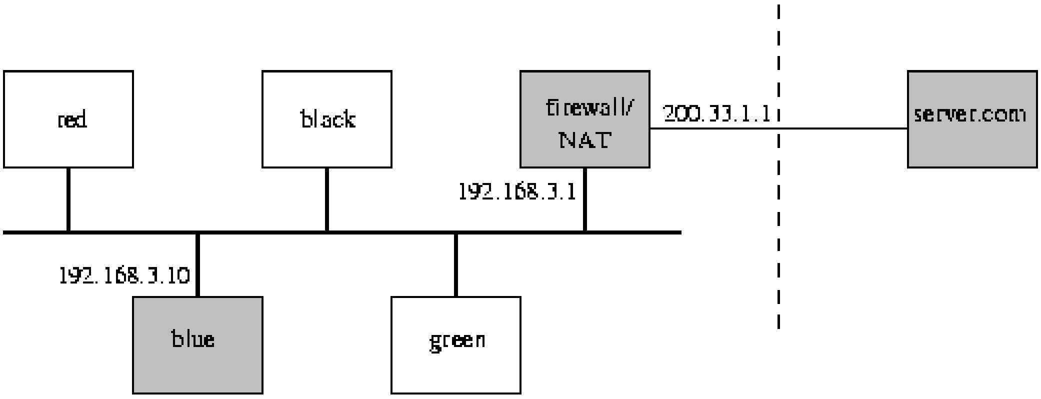
| beginning | ending | subnet-mask |
|---:|---:|---:|
| 10.0.0.1 | 10.255.255.254 | 10.0.0.0/8 |
| 172.16.0.1 | 172.31.255.254 | 172.16.0.0/12 |
| 192.168.0.1 | 192.168.255.254 | 192.168.0.0/16 |

The primary motivations for using NAT are:

- your network provider may only provide you with a single IP address to use - NAT permits multiple hosts to use the same IP address,
- it simplifies the later growth and re-design of a network, and
- external attackers cannot (easily) learn the topology of your internal network unless they penetrate your firewall.

## An Example of IP Masquerading

Consider the following example: Machine *blue* (with a single Ethernet interface, and IP address 192.168.3.10) generates a packet, *from* its port 400, destined for *server.com*.




When the packet arrives at the NAT-enabled firewall (on its *internal* Ethernet interface, IP address 192.168.3.1), the firewall will de-encapsulate the packet, and rewrite it so that it *appears* to have now originated *from* the *firewall* itself (with IP address 200.33.1.1, and a currently unused port on the firewall, 1430). The packet is finally *forwarded* on the *external* Ethernet interface.

| SRC computer | SRC IP | SRC port | Firewall's IP | Firewall's assigned port |
|---|---|---|---|---|
| blue | 192.168.3.10 | 400 | 200.33.1.1 | 1430 |
| black | 192.168.3.22 | 1814 | 200.33.1.1 | 1892 |
| red | 192.168.3.18 | 550 | 200.33.1.1 | 1434 |
| blue | 192.168.3.10 | 4412 | 200.33.1.1 | 1890 |
| green | 192.168.3.19 | 2410 | 200.33.1.1 | 1435 |

When a reply is received from *server.com*, its destination IP address will be 200.33.1.1, port 1430.

The firewall's mapping table is consulted to reverse the translation, changing the IP address to 192.168.3.10 (for *blue*), port 400.

## Network Address Translation (NAT)

NAT, as described in RFC1631, has many forms:

- *overloaded NAT* - maps multiple unroutable IP addresses to a single registered (routable) IP address by using different ports (as just seen). This is variously known as PAT (Port Address Translation), single address NAT or port-level multiplexed NAT,
- *dynamic NAT* - maps an unroutable IP address to one of a managed group of registered IP addresses, and
- *static NAT* - maps an unroutable IP address to a registered IP address on a one-to-one basis. This is required when a device needs to be accessible from outside the network, such as a web- or FTP-server.

## Supporting NAT with *iptables*

*iptables* supports NAT very simply. Consider a home system with *appp* external connection, and an Ethernet internal connection:

```
EXT=ppp0
PPP_IP=130.95.44.44

iptables -t nat -P PREROUTING  DROP
iptables -t nat -P POSTROUTING DROP
iptables        -P FORWARD     DROP

# NAT everything heading out the external interface
iptables -t nat -A POSTROUTING -s 192.168.1.0/24 \
          -o $EXT -j SNAT --to-source $PPP_IP

#This enables ip forwarding, and thus by extension, NAT
echo 1 > /proc/sys/net/ipv4/ip_forward
```

## Connection Tracking

Connection tracking refers to the ability for a firewall to maintain state information about connections - source and destination IP address and port number pairs (known as socket pairs), protocol types, connection state and timeouts.

Firewalls able to do this are termed *stateful*. Stateful firewalling is inherently more secure than its 'stateless' counterpart - the simple packet filtering commonly seen in most 'personal firewalls'.

Consider an candidate packet arriving on an external interface:

- if the packet matches an entry already recorded in the firewall's *state table*, the packet is part of an `ESTABLISHED` connection,
- if the packet is `ICMP` traffic it might be `RELATED` to a `UDP/TCP` connection already in the state table,
- the packet might be attempting to start a `NEW` connection, or
- it might be unrelated to any connection, we say `INVALID`.

To support connection-tracking for valid TCP traffic, in *iptables*, we employ the state-tracking *module*:

```
iptables -A INPUT  -p tcp -m state \
         --state ESTABLISHED -j ACCEPT
iptables -A OUTPUT -p tcp -m state \
         --state NEW,ESTABLISHED -j ACCEPT
```

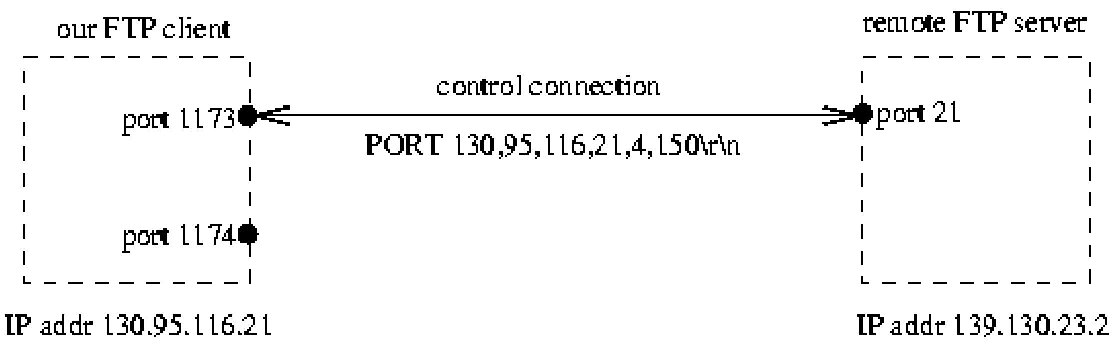
Under Linux we can see how many connections may be tracked from `/proc/sys/net/ipv4/ip_conntrack_max` (typically $2^{14}$), and can see the connections from `/proc/net/ip_conntrack`.

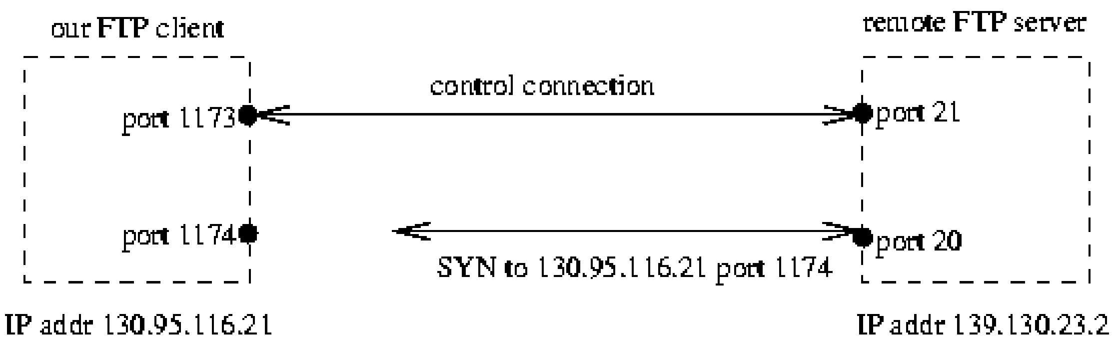
## Tracking *FTP* Connections

FTP transfers show the power of connection tracking. We can easily access a remote FTP service, and its *control-channel*:

```
iptables -A OUTPUT -p tcp --dport 21 -m state \
               --state NEW,ESTABLISHED -j ACCEPT
iptables -A INPUT  -p tcp --sport 21 -m state \
               --state ESTABLISHED -j ACCEPT
```

But that is not the whole story: we must also permit, seemingly 'random' connections to our FTP client's *data-port* as well.




Our FTP client sends its temporary port number over the FTP control-channel via a `PORT` command to the remote FTP server, which then connects from its port 20 to our specified port to send data, such as a file, or the output from a `DIR` request.

## Tracking *FTP* Connections, *continued*

To allow *active FTP* we may consider a general rule allowing connections from port 20 on remote FTP servers to high ports (port numbers > 1023) on our FTP clients.

However, this is too general to be considered secure, as remote attackers (who may be able to see our FTP `PORT` packets) may attempt to quickly connect to our nominated ports.

---

To solve this, (stateful firewalls, such as) *iptables* supports the specific *ip_conntrack_ftp* (dynamically loaded) module, which recognizes the `PORT` command and locates the port number (requiring parsing of the payload):

```
iptables -A INPUT  -p tcp --sport 20 -m state \
               --state ESTABLISHED,RELATED -j ACCEPT

iptables -A OUTPUT -p tcp --dport 20 -m state \
               --state ESTABLISHED -j ACCEPT
```

The FTP-data connection between our clients and the remote server is now classified as `RELATED` to the original outgoing connection to the remote port 21 - we don't need `NEW` as a state match.