Network Security

Security aspects of TCP/IP

Radboud University Nijmegen, The Netherlands

Autumn 2014

Collecting bits and pieces

0.0.0.0

- “This host on this network. MUST NOT be sent, except as a source address as part of an initialization procedure by which the host learns its own IP address.” (RFC 1122, Sec. 3.2.1.3 (a))
- Non-routable IP address
- Used for different purposes in different contexts:
  - IP address used for learning own IP address
  - Self-assigned invalid IP address
  - For (Python) server sockets: listen on any interface (but the empty string is portable to IPv6)

Collecting bits and pieces

0.0.0.0

- "This host on this network. MUST NOT be sent, except as a source address as part of an initialization procedure by which the host learns its own IP address." (RFC 1122, Sec. 3.2.1.3 (a))
- Non-routable IP address
- Used for different purposes in different contexts:
  - IP address used for learning own IP address
  - Self-assigned invalid IP address
  - For (Python) server sockets: listen on any interface (but the empty string is portable to IPv6)

Crossover cable

- Since Gigabit Ethernet we don't need crossover cables anymore
- Gigabit auto-detects inputs/outputs

A short recap

- Within the same subnet, it's fairly easy to sniff traffic
  - Hubs distribute data to everyone (but are largely obsolete)
  - Use ARP cache poisoning on switched Ethernet
  - Wireless LAN behaves a lot like hubbed Ethernet
- Protecting against ARP spoofing is hard: ARP does not support authentication

A short recap

- Within the same subnet, it's fairly easy to sniff traffic
  - Hubs distribute data to everyone (but are largely obsolete)
  - Use ARP cache poisoning on switched Ethernet
  - Wireless LAN behaves a lot like hubbed Ethernet
- Protecting against ARP spoofing is hard: ARP does not support authentication
- WiFi is generally easier to access than wired network (no physical protection)
- WiFi uses encryption to protect access to the network

A short recap

- Within the same subnet, it's fairly easy to sniff traffic
  - Hubs distribute data to everyone (but are largely obsolete)
  - Use ARP cache poisoning on switched Ethernet
  - Wireless LAN behaves a lot like hubbed Ethernet
- Protecting against ARP spoofing is hard: ARP does not support authentication
- WiFi is generally easier to access than wired network (no physical protection)
- WiFi uses encryption to protect access to the network
- WEP (Wired Equivalent Privacy) is broken (`aircrack-ng`)

A short recap

- Within the same subnet, it's fairly easy to sniff traffic
  - Hubs distribute data to everyone (but are largely obsolete)
  - Use ARP cache poisoning on switched Ethernet
  - Wireless LAN behaves a lot like hubbed Ethernet
- Protecting against ARP spoofing is hard: ARP does not support authentication
- WiFi is generally easier to access than wired network (no physical protection)
- WiFi uses encryption to protect access to the network
- WEP (Wired Equivalent Privacy) is broken (`aircrack-ng`)
- WPA (Wireless Protected Access) fixes some problems but still uses weak RC4 (TKIP)

A short recap

- Within the same subnet, it's fairly easy to sniff traffic
  - Hubs distribute data to everyone (but are largely obsolete)
  - Use ARP cache poisoning on switched Ethernet
  - Wireless LAN behaves a lot like hubbed Ethernet
- Protecting against ARP spoofing is hard: ARP does not support authentication
- WiFi is generally easier to access than wired network (no physical protection)
- WiFi uses encryption to protect access to the network
- WEP (Wired Equivalent Privacy) is broken (`aircrack-ng`)
- WPA (Wireless Protected Access) fixes some problems but still uses weak RC4 (TKIP)
- Problems properly solved in WPA2: uses AES (CCMP) instead of RC4

A short recap

- Within the same subnet, it's fairly easy to sniff traffic
  - Hubs distribute data to everyone (but are largely obsolete)
  - Use ARP cache poisoning on switched Ethernet
  - Wireless LAN behaves a lot like hubbed Ethernet
- Protecting against ARP spoofing is hard: ARP does not support authentication
- WiFi is generally easier to access than wired network (no physical protection)
- WiFi uses encryption to protect access to the network
- WEP (Wired Equivalent Privacy) is broken (`aircrack-ng`)
- WPA (Wireless Protected Access) fixes some problems but still uses weak RC4 (TKIP)
- Problems properly solved in WPA2: uses AES (CCMP) instead of RC4
- Most threatening in WPA2: bad passphrases, backwards-compatible TKIP

A short recap

- Within the same subnet, it's fairly easy to sniff traffic
  - Hubs distribute data to everyone (but are largely obsolete)
  - Use ARP cache poisoning on switched Ethernet
  - Wireless LAN behaves a lot like hubbed Ethernet
- Protecting against ARP spoofing is hard: ARP does not support authentication
- WiFi is generally easier to access than wired network (no physical protection)
- WiFi uses encryption to protect access to the network
- WEP (Wired Equivalent Privacy) is broken (`aircrack-ng`)
- WPA (Wireless Protected Access) fixes some problems but still uses weak RC4 (TKIP)
- Problems properly solved in WPA2: uses AES (CCMP) instead of RC4
- Most threatening in WPA2: bad passphrases, backwards-compatible TKIP
- Additional threat: WiFi Protected Setup (WPS)

RFCs

- RFC stands for *Request for Comments*
- Publications by the Internet Engineering Taskforce (IETF)
- Typically describes methods to operate the Internet
- Some RFCs are adopted as standards
- Published in plain ASCII
- Find them at `http://tools.ietf.org/rfc/rfcXXXX.txt`, where XXXX is the RFC number

RFCs

- RFC stands for *Request for Comments*
- Publications by the Internet Engineering Taskforce (IETF)
- Typically describes methods to operate the Internet
- Some RFCs are adopted as standards
- Published in plain ASCII
- Find them at `http://tools.ietf.org/rfc/rfcXXXX.txt`, where XXXX is the RFC number
- Not all RFCs are entirely serious, e.g., RFC 1149 "A Standard for the Transmission of IP Datagrams on Avian Carriers"

RFCs

- RFC stands for *Request for Comments*
- Publications by the Internet Engineering Taskforce (IETF)
- Typically describes methods to operate the Internet
- Some RFCs are adopted as standards
- Published in plain ASCII
- Find them at `http://tools.ietf.org/rfc/rfcXXXX.txt`, where XXXX is the RFC number
- Not all RFCs are entirely serious, e.g., RFC 1149 "A Standard for the Transmission of IP Datagrams on Avian Carriers"
- This RFC has actually been implemented, see `http://web-blug.rhcloud.com/rfc1149/writeup/`

From RFC 826 – An Ethernet Address Resolution Protocol

```
?Do I have the hardware type in ar$hrd?
Yes: (almost definitely)
  [optionally check the hardware length ar$hln]
  ?Do I speak the protocol in ar$pro?
  Yes:
    [optionally check the protocol length ar$pln]
    Merge_flag := false
    If the pair <protocol type, sender protocol address> is
        already in my translation table, update the sender
        hardware address field of the entry with the new
        information in the packet and set Merge_flag to true.
    ?Am I the target protocol address?
    Yes:
      If Merge_flag is false, add the triplet <protocol type,
          sender protocol address, sender hardware address> to
          the translation table.
      ?Is the opcode ares_op$REQUEST?  (NOW look at the opcode!!)
      Yes:
        Swap hardware and protocol fields, putting the local
            hardware and protocol addresses in the sender fields.
        Set the ar$op field to ares_op$REPLY
        Send the packet to the (new) target hardware address on
            the same hardware on which the request was received.
```

Another kind of ARP spoofing ctd.

- First a clarification of terms:
  - `ar$hrd`: Hardware address space, e.g., Ethernet
  - `ar$pro`: Protocol address space
  - `ar$hln`: byte length of each hardware address
  - `ar$pln`: byte length of each protocol address
  - `ares_op$REQUEST/REPLY`: Distinguish type (opcode)

Another kind of ARP spoofing ctd.

- First a clarification of terms:
  - `ar$hrd`: Hardware address space, e.g., Ethernet
  - `ar$pro`: Protocol address space
  - `ar$hln`: byte length of each hardware address
  - `ar$pln`: byte length of each protocol address
  - `ares_op$REQUEST/REPLY`: Distinguish type (opcode)
- Note: Update the table entry *before* checking REQUEST/REPLY
- Does not depend on gratuitous ARP
- Can use ARP requests to poison the ARP cache

Another kind of ARP spoofing ctd.

- First a clarification of terms:
  - `ar$hrd`: Hardware address space, e.g., Ethernet
  - `ar$pro`: Protocol address space
  - `ar$hln`: byte length of each hardware address
  - `ar$pln`: byte length of each protocol address
  - `ares_op$REQUEST/REPLY`: Distinguish type (opcode)
- Note: Update the table entry *before* checking REQUEST/REPLY
- Does not depend on gratuitous ARP
- Can use ARP requests to poison the ARP cache
- Can also create new entries in the ARP cache, not just overwrite existing ones

Another kind of ARP spoofing ctd.

- First a clarification of terms:
  - `ar$hrd`: Hardware address space, e.g., Ethernet
  - `ar$pro`: Protocol address space
  - `ar$hln`: byte length of each hardware address
  - `ar$pln`: byte length of each protocol address
  - `ares_op$REQUEST/REPLY`: Distinguish type (opcode)
- Note: Update the table entry *before* checking REQUEST/REPLY
- Does not depend on gratuitous ARP
- Can use ARP requests to poison the ARP cache
- Can also create new entries in the ARP cache, not just overwrite existing ones
- Ettercap supports this: set `arp_poison_request = 1` in `/etc/ettercap/etter.conf`

IP spoofing

- Generally: Send IP packet with wrong (“spoofed”) source address
- Easy to achieve with raw sockets (see first homework assignment)

IP spoofing

- Generally: Send IP packet with wrong (“spoofed”) source address
- Easy to achieve with raw sockets (see first homework assignment)
- Very problematic with services that use host-based authentication
- Classic example: `rlogin`, `rcp`, `rsh`
  - `rlogin`: Log into remote machine
  - `rsh`: Run command on remote machine
  - `rcp`: Copy files to remote machine

IP spoofing

- Generally: Send IP packet with wrong ("spoofed") source address
- Easy to achieve with raw sockets (see first homework assignment)
- Very problematic with services that use host-based authentication
- Classic example: `rlogin`, `rcp`, `rsh`
  - `rlogin`: Log into remote machine
  - `rsh`: Run command on remote machine
  - `rcp`: Copy files to remote machine
- No password required if the IP address is in `/etc/hosts.equiv` or `~/.rhosts`

IP spoofing

- Generally: Send IP packet with wrong ("spoofed") source address
- Easy to achieve with raw sockets (see first homework assignment)
- Very problematic with services that use host-based authentication
- Classic example: `rlogin`, `rcp`, `rsh`
  - `rlogin`: Log into remote machine
  - `rsh`: Run command on remote machine
  - `rcp`: Copy files to remote machine
- No password required if the IP address is in `/etc/hosts.equiv` or `~/.rhosts`
- This is to some extent fixed in recent versions of `rlogin`
- From the `rlogin` manpage: "the rlogin/rsh protocol in general, are inherently insecure and should be disabled if security is desired"

IP spoofing

- Generally: Send IP packet with wrong (“spoofed”) source address
- Easy to achieve with raw sockets (see first homework assignment)
- Very problematic with services that use host-based authentication
- Classic example: `rlogin`, `rcp`, `rsh`
  - `rlogin`: Log into remote machine
  - `rsh`: Run command on remote machine
  - `rcp`: Copy files to remote machine
- No password required if the IP address is in `/etc/hosts.equiv` or `~/.rhosts`
- This is to some extent fixed in recent versions of `rlogin`
- From the `rlogin` manpage: “the rlogin/rsh protocol in general, are inherently insecure and should be disabled if security is desired”
- IP spoofing is today mainly important in a larger attack context

SYN flooding

- Remember the three-way TCP handshake (SYN–SYN/ACK–ACK)
- After client sends SYN, server allocates resources for connection in the *SYN queue*, sends SYN/ACK

SYN flooding

- Remember the three-way TCP handshake (SYN–SYN/ACK–ACK)
- After client sends SYN, server allocates resources for connection in the *SYN queue*, sends SYN/ACK
- Attack: send many SYN packets, never reply to SYN/ACK with ACK
- Server's SYN queue will fill up
- DOS attack, preventing server from responding

SYN flooding

- Remember the three-way TCP handshake (SYN–SYN/ACK–ACK)
- After client sends SYN, server allocates resources for connection in the *SYN queue*, sends SYN/ACK
- Attack: send many SYN packets, never reply to SYN/ACK with ACK
- Server's SYN queue will fill up
- DOS attack, preventing server from responding
- Need to send SYN packets faster than server is "discarding" half-open connections
- Time to discard is configured by the TCP SYN-RECEIVED timer

SYN flooding

- Remember the three-way TCP handshake (SYN–SYN/ACK–ACK)
- After client sends SYN, server allocates resources for connection in the *SYN queue*, sends SYN/ACK
- Attack: send many SYN packets, never reply to SYN/ACK with ACK
- Server's SYN queue will fill up
- DOS attack, preventing server from responding
- Need to send SYN packets faster than server is "discarding" half-open connections
- Time to discard is configured by the TCP SYN-RECEIVED timer

Countermeasures

- Decrease the SYN-RECEIVED timer
- Increase the size of the queue
- Recycle oldest half-open connection
- Firewalls (later in this lecture)

SYN cookies

- SYN flooding countermeasures don't really solve the problem
- Better idea (Bernstein, Schenk, 1996): SYN cookies

SYN cookies

- SYN flooding countermeasures don't really solve the problem
- Better idea (Bernstein, Schenk, 1996): SYN cookies
- Reason for allocating resources after receiving SYN: need to remember properties of the connection
- Idea: Securely encode this information in the server's initial sequence number (ISN) of SYN/ACK
- Reconstruct information when ACK from client is received

SYN cookies

- SYN flooding countermeasures don't really solve the problem
- Better idea (Bernstein, Schenk, 1996): SYN cookies
- Reason for allocating resources after receiving SYN: need to remember properties of the connection
- Idea: Securely encode this information in the server's initial sequence number (ISN) of SYN/ACK
- Reconstruct information when ACK from client is received
- Compute ISN as the client's ISN plus offset of
  - top 5 bits: $t \mod 32$, where t is a 32-bit time counter that increases every 64 seconds
  - next 3 bits: an encoding of a maximal segment size (MSS) selected by the server in response to the client's MSS
  - bottom 24 bits: a server-selected secret function of the client IP address and port number, the server IP address and port number, and t.

SYN cookies ctd.

- When receiving an ACK,
  - check that the ACK number -1 corresponds to value of the secret function for a recent t
  - reconstruct the entry in the SYN queue
  - continue as usual

SYN cookies ctd.

- When receiving an ACK,
  - check that the ACK number -1 corresponds to value of the secret function for a recent t
  - reconstruct the entry in the SYN queue
  - continue as usual
- SYN cookies have some limitations (in particular TCP options), so only use them after the SYN queue is full

SYN cookies ctd.

- When receiving an ACK,
  - check that the ACK number -1 corresponds to value of the secret function for a recent t
  - reconstruct the entry in the SYN queue
  - continue as usual
- SYN cookies have some limitations (in particular TCP options), so only use them after the SYN queue is full
- Enable SYN cookies under Linux:
  ```
  echo 1 > /proc/sys/net/ipv4/tcp_syncookies
  ```

Ping of death

- DOS attack sometimes also possible by malformed packets
- Idea: target does not know how to handle packet and crashes

Ping of death

- DOS attack sometimes also possible by malformed packets
- Idea: target does not know how to handle packet and crashes
- Classical famous example: *ping of death*
- Idea is the following:
  - IP packets are limited to a length of 65535 bytes
  - IP packets get "chopped" into fragments for transportation through, e.g., Ethernet
  - IP header has a fragment offset
  - Fragment offset $+$ packet size must not exceed 65535

Ping of death

- DOS attack sometimes also possible by malformed packets
- Idea: target does not know how to handle packet and crashes
- Classical famous example: *ping of death*
- Idea is the following:
  - IP packets are limited to a length of 65535 bytes
  - IP packets get "chopped" into fragments for transportation through, e.g., Ethernet
  - IP header has a fragment offset
  - Fragment offset $+$ packet size must not exceed 65535
  - ... but it can
  - With fragmentation, it is possible to send IP packets of size > 65535

Ping of death

- DOS attack sometimes also possible by malformed packets
- Idea: target does not know how to handle packet and crashes
- Classical famous example: *ping of death*
- Idea is the following:
  - IP packets are limited to a length of 65535 bytes
  - IP packets get "chopped" into fragments for transportation through, e.g., Ethernet
  - IP header has a fragment offset
  - Fragment offset $+$ packet size must not exceed 65535
  - ... but it can
  - With fragmentation, it is possible to send IP packets of size > 65535
  - Receiving host will assemble the fragments into a buffer of size 65535

Ping of death

- DOS attack sometimes also possible by malformed packets
- Idea: target does not know how to handle packet and crashes
- Classical famous example: *ping of death*
- Idea is the following:
  - IP packets are limited to a length of 65535 bytes
  - IP packets get “chopped” into fragments for transportation through, e.g., Ethernet
  - IP header has a fragment offset
  - Fragment offset $+$ packet size must not exceed 65535
  - ... but it can
  - With fragmentation, it is possible to send IP packets of size > 65535
  - Receiving host will assemble the fragments into a buffer of size 65535
  - Overlong IP packet will overflow this buffer

Ping of death

- DOS attack sometimes also possible by malformed packets
- Idea: target does not know how to handle packet and crashes
- Classical famous example: *ping of death*
- Idea is the following:
  - IP packets are limited to a length of 65535 bytes
  - IP packets get "chopped" into fragments for transportation through, e.g., Ethernet
  - IP header has a fragment offset
  - Fragment offset $+$ packet size must not exceed 65535
  - ... but it can
  - With fragmentation, it is possible to send IP packets of size > 65535
  - Receiving host will assemble the fragments into a buffer of size 65535
  - Overlong IP packet will overflow this buffer
- This bug was present in UNIX, Linux, Windows, Mac, routers, printers ...
- Trivially easy to exploit with some implementations of `ping`:

```
ping -s 65510 target
```

ISN guessing

- Can we do more with IP spoofing than DOS?
- An man-in-the-middle attacker does not really need IP spoofing

ISN guessing

- Can we do more with IP spoofing than DOS?
- An man-in-the-middle attacker does not really need IP spoofing
- Problem without MitM (*off-path attack*):
  - TCP handshake SYN/ACK packet won't arrive at attacker
  - Attacker needs to generate valid ACK to establish connection
  - Valid ACK means: correct ACK number (server's ISN plus 1)
  - Can knock out the "real" receiver with DOS

ISN guessing

- Can we do more with IP spoofing than DOS?
- An man-in-the-middle attacker does not really need IP spoofing
- Problem without MitM (*off-path attack*):
  - TCP handshake SYN/ACK packet won't arrive at attacker
  - Attacker needs to generate valid ACK to establish connection
  - Valid ACK means: correct ACK number (server's ISN plus 1)
  - Can knock out the "real" receiver with DOS
- Can an attacker guess the server's ISN?

When new connections are created, an initial sequence number (ISN) generator is employed which selects a new 32 bit ISN. The generator is bound to a (possibly fictitious) 32 bit clock whose low order bit is incremented roughly every 4 microseconds. Thus, the ISN cycles approximately every 4.55 hours. Since we assume that segments will stay in the network no more than the Maximum Segment Lifetime (MSL) and that the MSL is less than 4.55 hours we can reasonably assume that ISN's will be unique." —RFC 793 (September 1981)

TCP SHOULD generate its Initial Sequence Numbers with the expression: ISN = M + F(localip, localport, remoteip, remoteport, secretkey) where M is the 4 microsecond timer, and F() is a pseudorandom function (PRF) of the connection-id. F() MUST NOT be computable from the outside, or an attacker could still guess at sequence numbers from the ISN used for some other connection. The PRF could be implemented as a cryptographic hash of the concatenation of the connection-id and some secret data; MD5 [RFC1321] would be a good choice for the hash function."

—RFC 6528 (February 2012)

. . . in the Linux kernel (3.16)

```
__u32 secure_tcp_sequence_number(__be32 saddr, __be32 daddr,
__be16 sport, __be16 dport)
{
  u32 hash[MD5_DIGEST_WORDS];

  net_secret_init();
  hash[0] = (__force u32)saddr;
  hash[1] = (__force u32)daddr;
  hash[2] = ((__force u16)sport << 16) + (__force u16)dport;
  hash[3] = net_secret[15];

  md5_transform(hash, net_secret);

  return seq_scale(hash[0]);
}
```

ISN guessing

- Can we do more with IP spoofing than DOS?
- An man-in-the-middle attacker does not really need IP spoofing
- Problem without MitM (*off-path attack*):
  - TCP handshake SYN/ACK packet won't arrive at attacker
  - Attacker needs to generate valid ACK to establish connection
  - Valid ACK means: correct ACK number (server's ISN plus 1)
- Can an attacker guess the server's ISN?
- Probably not easily (anymore)
- Keep in mind: No exact guess needed, attacker can try many sequence numbers!

TCP session hijacking

- Let's put all of this together, assume an off-track attacker
- Attack works as follows:

TCP session hijacking

- Let's put all of this together, assume an off-track attacker
- Attack works as follows:
  - Attacker launches DOS attack against A

TCP session hijacking

- Let's put all of this together, assume an off-track attacker
- Attack works as follows:
  - Attacker launches DOS attack against A
  - Attacker sends SYN packet to server B with source IP of A

TCP session hijacking

- Let's put all of this together, assume an off-track attacker
- Attack works as follows:
  - Attacker launches DOS attack against A
  - Attacker sends SYN packet to server B with source IP of A
  - Server sends SYN/ACK to A, but A does not reply (DOS)

TCP session hijacking

- Let's put all of this together, assume an off-track attacker
- Attack works as follows:
  - Attacker launches DOS attack against A
  - Attacker sends SYN packet to server B with source IP of A
  - Server sends SYN/ACK to A, but A does not reply (DOS)
  - Attacker sends ACK to B, guessing B's ISN

TCP session hijacking

- Let's put all of this together, assume an off-track attacker
- Attack works as follows:
  - Attacker launches DOS attack against A
  - Attacker sends SYN packet to server B with source IP of A
  - Server sends SYN/ACK to A, but A does not reply (DOS)
  - Attacker sends ACK to B, guessing B's ISN
  - Now B believes to have an established connection with A

TCP session hijacking

- Let's put all of this together, assume an off-track attacker
- Attack works as follows:
  - Attacker launches DOS attack against A
  - Attacker sends SYN packet to server B with source IP of A
  - Server sends SYN/ACK to A, but A does not reply (DOS)
  - Attacker sends ACK to B, guessing B's ISN
  - Now B believes to have an established connection with A
  - Attacker can now send packets through connection (but won't receive any)

TCP session hijacking

- Let's put all of this together, assume an off-track attacker
- Attack works as follows:
  - Attacker launches DOS attack against A
  - Attacker sends SYN packet to server B with source IP of A
  - Server sends SYN/ACK to A, but A does not reply (DOS)
  - Attacker sends ACK to B, guessing B's ISN
  - Now B believes to have an established connection with A
  - Attacker can now send packets through connection (but won't receive any)
- One-directional communication is enough to execute commands (e.g., `passwd`)

TCP session hijacking

- Let's put all of this together, assume an off-track attacker
- Attack works as follows:
  - Attacker launches DOS attack against A
  - Attacker sends SYN packet to server B with source IP of A
  - Server sends SYN/ACK to A, but A does not reply (DOS)
  - Attacker sends ACK to B, guessing B's ISN
  - Now B believes to have an established connection with A
  - Attacker can now send packets through connection (but won't receive any)
- One-directional communication is enough to execute commands (e.g., `passwd`)
- Attacker can also take over existing, legitimate connection between A and B

Ports and Services

- TCP/IP communication first needs a server to open a port
- "Speaking" to the service on the other side needs knowledge about the higher-level protocol

Ports and Services

- TCP/IP communication first needs a server to open a port
- "Speaking" to the service on the other side needs knowledge about the higher-level protocol
- Some services announce what they are through a "banner"

Ports and Services

- TCP/IP communication first needs a server to open a port
- "Speaking" to the service on the other side needs knowledge about the higher-level protocol
- Some services announce what they are through a "banner"
- *Internet Assigned Numbers Authority* (IANA) defines list of known ports and services
- Same port for UDP and TCP (but service is not necessarily listening on both)
- List in file `/etc/services`
- It is of course not mandatory to use these ports, but it's what clients assume

Common services and their ports

TCP/UDP port	Service
21	File Transfer Protocol (FTP)
22	Secure Shell (SSH)
25	Simple Mail Transfer Protocol (SMTP)
53	Domain Name Server
80	Hypertext Transfer Protocol (HTTP)
110	Post Office Protocol (POP3)
143	Interim Mail Access Protocol (IMAP)
443	HTTP over SSL/TLS (HTTPS)
465	SMTP over SSL/TLS (SMTPS)
993	IMAP over SSL/TLS (IMAPS)
995	POP3 over SSL/TLS (POP3S)

`netstat`

- Very important to know and understand: local listening programs/ports
- Various examples:
  - `netstat -tl`: All listening TCP ports
  - `netstat -ul`: All listening UDP ports
  - `netstat -al`: All listening ports (also UNIX ports)
- The `--program` option also shows which process opened the connection
- Run as root to see all `--program` information

telnet, netcat, and openssl

- Can use netcat to connect to any port:

```
netcat www.google.com 80
```

telnet, netcat, and openssl

- Can use `netcat` to connect to any port:

  `netcat www.google.com 80`

- Alternative: `telnet`, for example

  `telnet www.google.com 80`

- Originally made to provide command-line interface to remote host
- Telnet server's standard port is 23 (insecure and obsolete today)
- Can also be used to connect to any other port, behaves much like `netcat` (with small differences for line endings etc.)

telnet, netcat, and openssl

- Can use `netcat` to connect to any port:

  ```
  netcat www.google.com 80
  ```

- Alternative: `telnet`, for example

  ```
  telnet www.google.com 80
  ```

- Originally made to provide command-line interface to remote host
- Telnet server's standard port is 23 (insecure and obsolete today)
- Can also be used to connect to any other port, behaves much like `netcat` (with small differences for line endings etc.)
- `netcat` and `telnet` don't work with SSL connections
- Use OpenSSL's `s_client` instead, e.g.:

  ```
  openssl s_client -connect encrypted.google.com:443
  ```

Port scanning – `nmap`

- Typical thing to first figure out about a remote, unknown computer: list of open ports
- *Port scanning* means "trying all ports"

Port scanning – `nmap`

- Typical thing to first figure out about a remote, unknown computer: list of open ports
- *Port scanning* means "trying all ports"
- Widely used tool for port scans: `nmap`
- A simple `nmap arya` will scan 1000 ports on `arya`

Port scanning – `nmap`

- Typical thing to first figure out about a remote, unknown computer: list of open ports
- *Port scanning* means "trying all ports"
- Widely used tool for port scans: `nmap`
- A simple `nmap arya` will scan 1000 ports on `arya`
- Default scan method for non-privileged user: `connect()` scan:
  - Use the OS's `connect()` system call to connect to a remote port
  - `connect()` succeeds: port is open
  - `connect()` fails: port is closed
  - Immediately close connection after successful `connect()`
  - This scanning method does not need root privileges

Port scanning – `nmap`

- Typical thing to first figure out about a remote, unknown computer: list of open ports
- *Port scanning* means “trying all ports”
- Widely used tool for port scans: `nmap`
- A simple `nmap arya` will scan 1000 ports on `arya`
- Default scan method for non-privileged user: `connect()` scan:
  - Use the OS’s `connect()` system call to connect to a remote port
  - `connect()` succeeds: port is open
  - `connect()` fails: port is closed
  - Immediately close connection after successful `connect()`
  - This scanning method does not need root privileges
- “Filtered” means that a firewall blocks access (more later in this lecture)

Port scanning – `nmap`

- Typical thing to first figure out about a remote, unknown computer: list of open ports
- *Port scanning* means "trying all ports"
- Widely used tool for port scans: `nmap`
- A simple `nmap arya` will scan 1000 ports on `arya`
- Default scan method for non-privileged user: `connect()` scan:
  - Use the OS's `connect()` system call to connect to a remote port
  - `connect()` succeeds: port is open
  - `connect()` fails: port is closed
  - Immediately close connection after successful `connect()`
  - This scanning method does not need root privileges
- "Filtered" means that a firewall blocks access (more later in this lecture)
- Scan all ports (including high ports) through

```
nmap -p 1-65535 arya
```

SYN, Null, FIN and Xmas scans

- `connect()` scans appear in the servers' log files
- Sometimes a more "stealthy" scan is desired
- Only need a "distinguisher" between open and closed ports

SYN, Null, FIN and Xmas scans

SYN scan

- Send SYN packet
- Receiving SYN/ACK: port is open
- Receiving RST: port is closed
- Send an RST when receiving SYN/ACK to "hang up"
- Connection is never completed (service does not log it)
- Default in nmap with root privileges (or use `-sS`)

SYN, Null, FIN and Xmas scans

Null, FIN, and Xmas scans

- RFC 793 states in Section 3.9:
  - "If the port state is CLOSED . . . An incoming segment not containing a RST causes a RST to be sent in response."

SYN, Null, FIN and Xmas scans

Null, FIN, and Xmas scans

- RFC 793 states in Section 3.9:
  - "If the port state is CLOSED ... An incoming segment not containing a RST causes a RST to be sent in response."
  - "If the state is LISTEN ... Any other control or text-bearing segment (not containing SYN) must have an ACK and thus would be discarded by the ACK processing. An incoming RST segment could not be valid, since it could not have been sent in response to anything sent by this incarnation of the connection. So you are unlikely to get here, but if you do, drop the segment, and return."

SYN, Null, FIN and Xmas scans

Null, FIN, and Xmas scans

- RFC 793 states in Section 3.9:
  - "If the port state is CLOSED ... An incoming segment not containing a RST causes a RST to be sent in response."
  - "If the state is LISTEN ... Any other control or text-bearing segment (not containing SYN) must have an ACK and thus would be discarded by the ACK processing. An incoming RST segment could not be valid, since it could not have been sent in response to anything sent by this incarnation of the connection. So you are unlikely to get here, but if you do, drop the segment, and return."
- Any packet without SYN,ACK, or RST can serve as distinguisher
- Null scan: no flags set (`-sN`)
- FIN scan: FIN flag set (`-sF`)
- Xmas scan: FIN,PSH, and URG flag set (`-sX`)

SYN, Null, FIN and Xmas scans

Null, FIN, and Xmas scans

- RFC 793 states in Section 3.9:
  - "If the port state is CLOSED ... An incoming segment not containing a RST causes a RST to be sent in response."
  - "If the state is LISTEN ... Any other control or text-bearing segment (not containing SYN) must have an ACK and thus would be discarded by the ACK processing. An incoming RST segment could not be valid, since it could not have been sent in response to anything sent by this incarnation of the connection. So you are unlikely to get here, but if you do, drop the segment, and return."
- Any packet without SYN,ACK, or RST can serve as distinguisher
- Null scan: no flags set (`-sN`)
- FIN scan: FIN flag set (`-sF`)
- Xmas scan: FIN,PSH, and URG flag set (`-sX`)
- Problem: Not all operating systems behave according to RFC 793
- For example, Windows will always send RST (making all ports look closed)

Idle scans

- With TCP SYN and FIN scans there is still bidirectional communication between scanning and scanned host
- More "stealthy": *idle scan* using a *zombie host*

Idle scans

- With TCP SYN and FIN scans there is still bidirectional communication between scanning and scanned host
- More “stealthy”: *idle scan* using a *zombie host*
- Idle scans use the following features:
  - A SYN will be answered by SYN/ACK (open) or RST (closed)
  - An unsolicited SYN/ACK is answered by RST
  - An unsolicited RST is dropped

Idle scans

- With TCP SYN and FIN scans there is still bidirectional communication between scanning and scanned host
- More "stealthy": *idle scan* using a *zombie host*
- Idle scans use the following features:
  - A SYN will be answered by SYN/ACK (open) or RST (closed)
  - An unsolicited SYN/ACK is answered by RST
  - An unsolicited RST is dropped
  - IP packets contain the *fragment identification number* (IPID)
  - Many OS increase the IPID by one for every packet sent

Idle scans

- With TCP SYN and FIN scans there is still bidirectional communication between scanning and scanned host
- More "stealthy": *idle scan* using a *zombie host*
- Idle scans use the following features:
  - A SYN will be answered by SYN/ACK (open) or RST (closed)
  - An unsolicited SYN/ACK is answered by RST
  - An unsolicited RST is dropped
  - IP packets contain the *fragment identification number* (IPID)
  - Many OS increase the IPID by one for every packet sent
- A zombie host is an idle machine on the network
- Idle scan proceeds as follows:
  - Probe the zombie's IPID and record it, let's say IPID$= X$
  - Forge SYN packet from the zombie to the target host and port
  - Probe the zombie's IPID again, let say IPID$= Y$
  - $Y = X + 1$: port is closed
  - $Y = X + 2$: port is open

Idle scans

- With TCP SYN and FIN scans there is still bidirectional communication between scanning and scanned host
- More "stealthy": *idle scan* using a *zombie host*
- Idle scans use the following features:
  - A SYN will be answered by SYN/ACK (open) or RST (closed)
  - An unsolicited SYN/ACK is answered by RST
  - An unsolicited RST is dropped
  - IP packets contain the *fragment identification number* (IPID)
  - Many OS increase the IPID by one for every packet sent
- A zombie host is an idle machine on the network
- Idle scan proceeds as follows:
  - Probe the zombie's IPID and record it, let's say IPID$= X$
  - Forge SYN packet from the zombie to the target host and port
  - Probe the zombie's IPID again, let say IPID$= Y$
  - $Y = X + 1$: port is closed
  - $Y = X + 2$: port is open
- Can use, for example, a printer as zombie

Idle scans

- With TCP SYN and FIN scans there is still bidirectional communication between scanning and scanned host
- More "stealthy": *idle scan* using a *zombie host*
- Idle scans use the following features:
  - A SYN will be answered by SYN/ACK (open) or RST (closed)
  - An unsolicited SYN/ACK is answered by RST
  - An unsolicited RST is dropped
  - IP packets contain the *fragment identification number* (IPID)
  - Many OS increase the IPID by one for every packet sent
- A zombie host is an idle machine on the network
- Idle scan proceeds as follows:
  - Probe the zombie's IPID and record it, let's say IPID$= X$
  - Forge SYN packet from the zombie to the target host and port
  - Probe the zombie's IPID again, let say IPID$= Y$
  - $Y = X + 1$: port is closed
  - $Y = X + 2$: port is open
- Can use, for example, a printer as zombie
- Idle scan with `nmap`: `nmap -sI zombie`

How about UDP?

- UDP is stateless, how do you scan UDP ports?

How about UDP?

- UDP is stateless, how do you scan UDP ports?
- First option: Use ICMP port unreachable:
  - No response means: port is open (or filtered)
  - ICMP port unreachable means: port is closed

How about UDP?

- UDP is stateless, how do you scan UDP ports?
- First option: Use ICMP port unreachable:
  - No response means: port is open (or filtered)
  - ICMP port unreachable means: port is closed
- Problem: rate limitation for ICMP port unreachable

How about UDP?

- UDP is stateless, how do you scan UDP ports?
- First option: Use ICMP port unreachable:
  - No response means: port is open (or filtered)
  - ICMP port unreachable means: port is closed
- Problem: rate limitation for ICMP port unreachable
- Second option: Use specific services, for example:
  - DNS uses UDP on port 53
  - Send DNS request to UDP port 53
  - Getting a DNS reply back means that there is a DNS server

How about UDP?

- UDP is stateless, how do you scan UDP ports?
- First option: Use ICMP port unreachable:
  - No response means: port is open (or filtered)
  - ICMP port unreachable means: port is closed
- Problem: rate limitation for ICMP port unreachable
- Second option: Use specific services, for example:
  - DNS uses UDP on port 53
  - Send DNS request to UDP port 53
  - Getting a DNS reply back means that there is a DNS server
- UDP scans in `nmap`: `nmap -sU`

OS fingerprinting

- Important information about target host/network: OS
- TCP/IP leaves details of various parameters to the implementation
- Different operating systems use different parameters
- Investigating those parameters gives information about OS
- TCP/IP fingerprinting with `nmap`: `nmap -O`

OS fingerprinting

- Important information about target host/network: OS
- TCP/IP leaves details of various parameters to the implementation
- Different operating systems use different parameters
- Investigating those parameters gives information about OS
- TCP/IP fingerprinting with `nmap`: `nmap -O`
- Another way to detect OS: "talk" to open ports
- Many services reveal details (e.g., banner information)
- Knowing that Microsoft IIS runs on port 80 say a lot about the OS
- Run `nmap -sV` for `version detection`

OS fingerprinting

- Important information about target host/network: OS
- TCP/IP leaves details of various parameters to the implementation
- Different operating systems use different parameters
- Investigating those parameters gives information about OS
- TCP/IP fingerprinting with `nmap`: `nmap -O`
- Another way to detect OS: "talk" to open ports
- Many services reveal details (e.g., banner information)
- Knowing that Microsoft IIS runs on port 80 say a lot about the OS
- Run `nmap -sV` for `version detection`
- Convenient shortcut: `nmap -A` (`-O -sV -sC --traceroute`)

Portscans – attack or not?

Port scans: no attack

- You only look for offered services
- If you don't want a service to be found, don't offer that service
- Port scans are important tools for administrators to verify security policies
- Blocking port-scans through firewalls can easily break other functionality

Portscans – attack or not?

Port scans: no attack

- You only look for offered services
- If you don't want a service to be found, don't offer that service
- Port scans are important tools for administrators to verify security policies
- Blocking port-scans through firewalls can easily break other functionality

Port scans – (part of) an attack

- Why would I want to reveal more about my system than I have to?
- Port scans are a typical first step of an attack
- "If I want you to know about an open service, I'll tell you"
- `nmap` manpage gives a few hints...:

```
peter@tyrion: $ man nmap | grep -o attack | wc -l
18
```

NSA/GCHQ Project Hacienda

- August 2014: Leak about the NSA/GCHQ Hacienda program
- Port scan entire nations (27 completed, 5 partially completed) using `nmap`
- Port scanning (reconnaissance) first step of a 4-step process:
  - Reconnaissance
  - Infection
  - Command and Control
  - Exfiltration
- Automate the process of analyzing `nmap` data (project OLYMPIA)
- Take control over vulnerable hosts and turn them into Operational Relay Boxes (ORBs)
- For more details, see http://www.heise.de/ct/artikel/NSA-GCHQ-The-HACIENDA-Program-for-Internet-Colonization-2292681.html

Portknocking

- Some services are meant to be public, e.g., HTTP(S), SMTP(S)
- Other services are (often) only meant for one or very few users, e.g., SSH
- Can run those service on non-standard ports (e.g., 51966): security by obscurity

Portknocking

- Some services are meant to be public, e.g., HTTP(S), SMTP(S)
- Other services are (often) only meant for one or very few users, e.g., SSH
- Can run those service on non-standard ports (e.g., 51966): security by obscurity
- Idea: "Hide" those services, only open them for intended user

Portknocking

- Some services are meant to be public, e.g., HTTP(S), SMTP(S)
- Other services are (often) only meant for one or very few users, e.g., SSH
- Can run those service on non-standard ports (e.g., 51966): security by obscurity
- Idea: “Hide” those services, only open them for intended user
- Wait for certain packets to arrive, then open port (or firewall rule)
- This concept is called *port knocking*
- Sequence of special packets is called *knocking sequence*

Portknocking

- Some services are meant to be public, e.g., HTTP(S), SMTP(S)
- Other services are (often) only meant for one or very few users, e.g., SSH
- Can run those service on non-standard ports (e.g., 51966): security by obscurity
- Idea: "Hide" those services, only open them for intended user
- Wait for certain packets to arrive, then open port (or firewall rule)
- This concept is called *port knocking*
- Sequence of special packets is called *knocking sequence*
- Simple example:
  - Send UDP packets to ports 42, 53, 4000, 666 from IP 1.2.3.4
  - This opens port 22 (SSH) for connections from IP 1.2.3.4

Portknocking

- Some services are meant to be public, e.g., HTTP(S), SMTP(S)
- Other services are (often) only meant for one or very few users, e.g., SSH
- Can run those service on non-standard ports (e.g., 51966): security by obscurity
- Idea: "Hide" those services, only open them for intended user
- Wait for certain packets to arrive, then open port (or firewall rule)
- This concept is called *port knocking*
- Sequence of special packets is called *knocking sequence*
- Simple example:
  - Send UDP packets to ports 42, 53, 4000, 666 from IP 1.2.3.4
  - This opens port 22 (SSH) for connections from IP 1.2.3.4
- Port scanners won't see port 22 as open
- Can still use SSH from anywhere (if you know the knocking sequence)

More portknocking

- Various ways to implement port knocking:
  - Kernel space vs. user space
  - TCP vs. UDP
  - Inspecting every packet with libpcap vs. lightweight methods (e.g., logfiles)
  - Multi-packet vs. single-packet (Single Packet Authorization (SPA))
  - Protection against replay attacks
  - Cryptographic protection and authentication

More portknocking

- Various ways to implement port knocking:
  - Kernel space vs. user space
  - TCP vs. UDP
  - Inspecting every packet with libpcap vs. lightweight methods (e.g., logfiles)
  - Multi-packet vs. single-packet (Single Packet Authorization (SPA))
  - Protection against replay attacks
  - Cryptographic protection and authentication
- Nice summary of the reason for port knocking: *"Because you are running network services with security vulnerabilities in them. Again, you are running network services with security vulnerabilities in them. If you're running a server, this is almost universally true. Most software is complex. It changes rapidly, and innovation tends to make it more complex. It is going to be, forever, hopelessly, insecure."* —Moxie Marlinspike

TCP Stealth

- Recent proposal by Kirsch and Grothoff (August 2014): TCP Stealth
- No additional packets for port knocking
- Idea: Include authentication in client's ISN

TCP Stealth

- Recent proposal by Kirsch and Grothoff (August 2014): TCP Stealth
- No additional packets for port knocking
- Idea: Include authentication in client's ISN
- TCP traffic indistiguishable from "normal" traffic for passive attacker
- Relatively low cryptographic security (2^{32})

TCP Stealth

- Recent proposal by Kirsch and Grothoff (August 2014): TCP Stealth
- No additional packets for port knocking
- Idea: Include authentication in client's ISN
- TCP traffic indistiguishable from "normal" traffic for passive attacker
- Relatively low cryptographic security (2^{32})
- Integrity protection of first TCP segment
- Idea: if this segment contains public-key material, shut out a MitM

TCP Stealth

- Recent proposal by Kirsch and Grothoff (August 2014): TCP Stealth
- No additional packets for port knocking
- Idea: Include authentication in client's ISN
- TCP traffic indistiguishable from "normal" traffic for passive attacker
- Relatively low cryptographic security (2^{32})
- Integrity protection of first TCP segment
- Idea: if this segment contains public-key material, shut out a MitM
- Compatible with SYN cookies

TCP Stealth

- Recent proposal by Kirsch and Grothoff (August 2014): TCP Stealth
- No additional packets for port knocking
- Idea: Include authentication in client's ISN
- TCP traffic indistiguishable from "normal" traffic for passive attacker
- Relatively low cryptographic security (2^{32})
- Integrity protection of first TCP segment
- Idea: if this segment contains public-key material, shut out a MitM
- Compatible with SYN cookies
- Implemented for Linux kernel
- Submitted as an IETF draft: https://datatracker.ietf.org/doc/draft-kirsch-ietf-tcp-stealth/

TCP Stealth

- Recent proposal by Kirsch and Grothoff (August 2014): TCP Stealth
- No additional packets for port knocking
- Idea: Include authentication in client's ISN
- TCP traffic indistiguishable from "normal" traffic for passive attacker
- Relatively low cryptographic security (2^{32})
- Integrity protection of first TCP segment
- Idea: if this segment contains public-key material, shut out a MitM
- Compatible with SYN cookies
- Implemented for Linux kernel
- Submitted as an IETF draft: https://datatracker.ietf.org/doc/draft-kirsch-ietf-tcp-stealth/
- For more details, see https://gnunet.org/kirsch2014knock

See you next week

Image source: http://www.costao.com.br