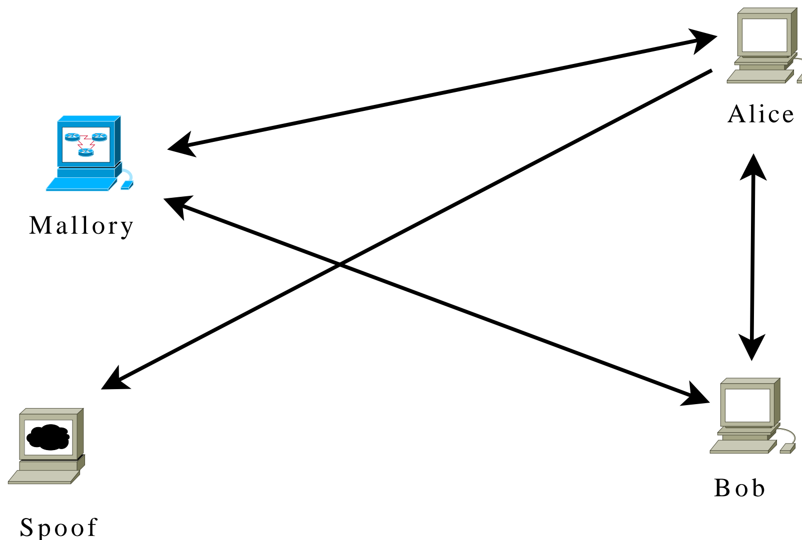
Einleitung

- Angriff von Kevin Mitnick auf Rechner von Tsutomu Shimomura Weihnachten 1994.
- Tsutomu Shimomura war Spezialist für Rechnersicherheit bei SUN Microsystems
- John Markoff, damals Journalist der New York Times hat den Angriff bekannt gemacht.
- Die Details zum Angriff sind z.B. hier dokumentiert `http://www.gulker.com/ra/hack/`

Beteiligte Systeme



Finger und Stealth SYN Scan

- **mallory**# finger -l @**alice**
  Liste der Benutzer auf **alice**
- **mallory**# finger -l @**bob**
  Liste der Benutzer auf **bob**
- **mallory**# finger -l [2]**root@bob**
  Details einer Session des Benutzers **root** auf **bob**, z.B.:
  Last login Thu Aug 5 09:55 (PDT) on ttyx1 from **alice**
- 1 mallory.51916 > alice.login: S
  2 alice.login > mallory.51916: S ack
  Test, ob auf **alice** der `rlogin` Dienst aktiviert ist

TCP Sequenznummern

- 1 mallory.1000 > bob.shell: S 1382726990(0)
  2 bob.shell > mallory.1000: S 2021824000(0) ack 1382726991
  3 mallory.1000 > bob.shell: R 1382726991(0)
- 1 mallory.999 > bob.shell: S 1382726991(0)
  2 bob.shell > mallory.999: S 2021952000(0) ack 1382726992
  3 mallory.999 > bob.shell: R 1382726992(0)
- 1 mallory.998 > bob.shell: S 1382726992(0)
  2 bob.shell > mallory.998: S 2022080000(0) ack 1382726993
  3 mallory.998 > bob.shell: R 1382726993(0)
- Die initialen Sequenznummern von **bob** wachsen pro Verbindungsaufbau um 128000

SYN Flood Angriff auf **alice.login**

- 1 spoof.600 > alice.login: S 1382726960(0)
  2 alice.login > spoof.600: S 1022085000(0) ack 1382726961
  3 spoof.601 > alice.login: S 1382726961(0)
  4 alice.login > spoof.601: S 1022086000(0) ack 1382726962
  ...
  15 spoof.607 > alice.login: S 1382726967
  16 alice.login > spoof.607: S 1022092000(0) ack 1382726968
  Bis hier wurden von **alice** SYN ACK erzeugt
- 17 spoof.608 > alice.login: S 1382726968(0)
  18 spoof.609 > alice.login: S 1382726969(0)

  Die Listen-Queue von **alice** ist voll, SYN ACK Pakete werden verworfen, sofern sie nicht ein gespeichertes SYN beantworten.

IP Spoofing Angriff auf **bob.login**

- **mallory** schickt ein SYN Paket mit falscher Quell-IP an **bob**
  1 alice.login > bob.shell: S 1382727010:1382727010(0)
- **bob** antwortet **alice** 2 bob.shell > alice.login: S 2022208000(0) ack 1382727011
- **alice** verwirft den SYN ACK, sendet kein RST
- **mallory** sieht den SYN ACK nicht, kann den Inhalt aber vorhersagen
- **mallory** sendet ACK, damit ist die Verbindung ESTABLISHED
  3 alice.login > bob.shell: . ack 2022208001
- **mallory** sendet Schadcode
  Daten: ..root.root.echo + + » /.rhosts
  5 alice.login > bob.shell: F 1382727044(0) ack 2022208001

Konsequenzen

- rlogin -l root bob funktioniert ohne Authentifizierung aus dem Internet
- Kevin Mitnick wurde später zu 46 Monaten Haft verurteilt
- Moderne Betriebssysteme benutzen zufällige initiale Sequenznummern
- rlogin/rsh/rexec sind durch ssh weitgehend abgelöst
- SYN Flooding bleibt weiter ein Problem, SYN Cookies nur ein Behelf

Grundlagen

- Wert des Protokoll Feldes im IP Header ist 17 (vgl. RFC1700)
- Jedes UDP Datagramm erzeugt genau ein (möglicherweise fragmentiertes) IP Paket
- ungesicherter, verbindungsloser Transport
- UDP unterstützt Unicast und Multicast
- Daten sind bei UDP immer eine Folge von 8 Bit Zeichen

UDP Rahmen

Source Port	Destination Port
UDP Length	UDP Checksum
UDP Data	

Felder im UDP Header

- **Source Port:** Nummer, mit der die sendende Applikation vom Host identifiziert werden kann.
- **Destination Port:** Nummer, mit der die empfangende Applikation vom Host identifiziert werden kann.
- **UDP Length:** 16 Bit Länge des gesamten Datagramms in Bytes
- **Checksum:** 16 Bit Prüfsumme des gesamten Datagramms zuzüglich eines Pseudoheaders

UDP Prüfsummenberechnung

Dem UDP Datagramm wird ein Pseudoheader vorangestellt, bei ungerader Anzahl Bytes mit einer 0 aufgefüllt und dann das für IP üblicher Verfahren angewendet. Ist das Ergebnis 0, wird 0*xFFFF* eingesetzt, ist das Feld 0, wurde es nicht berechnet.

<table>
<tr><td colspan="3">Source IP Address</td></tr>
<tr><td colspan="3">Destination IP Address</td></tr>
<tr><td>0</td><td>Protocol</td><td>UDP Length</td></tr>
<tr><td colspan="2">Source Port</td><td>Destination Port</td></tr>
<tr><td colspan="2">UDP Length</td><td>UDP Checksum</td></tr>
<tr><td colspan="3">(Padded) UDP Data</td></tr>
</table>

Größe von UDP Datagrammen

- Der UDP Header erlaubt 65507 Bytes Daten (8 Byte UDP Header, 20 Byte IP Header)
- Ein Host muß IP Pakete von mindestens 576 Byte empfangen können (RFC791), d.h. mindestend 548 Byte UDP Daten.
- Liest eine Anwendung erfolgreich empfangene Daten, kann das Datagramm gekürzt werden.
- Der Zwischenspeicher kann von der Anwendungsschicht vergrößert oder verkleinert werden.

Die meisten UDP basierten Protokolle gehen davon aus, daß maximal 512 Byte Daten pro Datagramm übertragen und verarbeitet werden können.

Standard

- Entstanden an der Berkeley University of California
- Veröffentlicht als Programmierschnittstelle des BSD4.2 (1983)
- Inzwischen standardisiert als IEEE Std. 1003.1-2004, als Single Unix Specification der Open Group und als ISO/IEC 9945:2002
- Die standardisierte C Schnittstelle ist aus verschiedenen Programmiersprachen nutzbar.

Umfang

Durch den Standard werden Schnittstellen beschrieben, die folgende Aspekte des Zugriffs auf die Transportschicht regeln:

- Adressierung, Adressumwandlung
- Verbindungsaufbau/Verbindungsabbau
- Konfiguration der Interna einer Verbindung
- Konfiguration der möglichen Zugriffsmethoden
- Datentransfer

Adressierung

Die BSD Socket API bietet Datenstrukturen und Konvertierungsfunktionen für Adressen verschiedener Formate:

- `inet_ntop, inet_pton`: Umwandlung von IP Adressen in binäre Darstellung (IPv4 und IPv6)
- `getaddrinfo`: Umwandlung eines symbolischen Namens in Adresse(n) der Transportschicht
- `getpeername`: Liefert die Adresse der Gegenseite einer Verbindung

Verbindungsaufbau/Verbindungsabbau

- `socket`: Belegt Resource (sog. Socket) für den Zugriff auf die Verbindung
- `bind`: Setzt Quelladresse der Verbindung
- `connect`: Setzt Zieladresse und baut ggfs. Verbindung auf (vgl. active open)
- `listen`, `accept`: Läßt Socket auf Verbindungsaufbau warten (vgl. passive open)
- `shutdown`: Schließt eine Richtung einer Verbindung
- `close`: Schließt eine Verbindung

Datentransfer

- `select, poll`: Testet, ob ein Ereignis auf dem Socket wartet.
- `send, sendto, sendmsg`: Sende Daten, abhängig vom gewählten Transport
- `recv, recvfrom, recvmsg`: Empfange Daten, abhängig vom gewählten Transport

Konfiguration

Die Konfiguration der Vermittlungsschicht und Transportschicht erfolgt über `setsockopt` und `fcntl`q. Es folgt eine Auswahl der portablen Optionen:

- `SO_BROADCAST`: Erlaube Broadcasts bei UDP
- `SO_ERROR`: Lese Fehler mit `getsockopt`
- `SO_KEEPALIVE`: Schalte TCP Keepalive ein
- `SO_LINGER`: Warte, bis alle Daten bei einem `close` oder `shutdown` erfolgreich übertragen sind
- `SO_RCVBUF`: Größe des Empfanspuffers
- `SO_RCVLOWAT`: Mindestmenge an Daten, die zur Anwendung hochgereicht werden
- `SO_RCVTIMEO`: Timeout beim Lesen von Daten aus dem Empfangspuffer

Konfiguration (2)

- SO_REUSEADDR: Verwende Adresse neu, ohne 2 MSL Timeout abzuwarten
- SO_SNDBUF: Größe des Sendepuffers
- SO_SNDLOWAT: Minimale Datenmenge bei Sendeoperationen
- SO_SNDTIMEO: Timeout beim Senden
- SO_TYPE: Abfrage des Verbindungstype/Protokolls
-
- TCP_MAXSEG: Setze MSS
- TCP_NODELAY: Schaltet Nagle Algoritmus aus
- O_NONBLOCK: Aufrufe auf dem Socket blockieren nicht mehr.

Client

```
from socket import *
from ReadUntilEOF import *

c_so = socket(AF_INET, SOCK_STREAM)
c_so.setsockopt(SOL_SOCKET, SO_REUSEADDR,1)
c_so.bind( ( "0.0.0.0", 7777 ) )

c_so.connect( ( "localhost", 6666 ) )

c_so.sendall("client request line 1\n")
c_so.sendall("client request line 2\n")
c_so.shutdown(SHUT_WR)

print readUntilEOF(c_so)

c_so.close()
```

Server

```
from socket import *
from ReadUntilEOF import *

a_so = socket(AF_INET, SOCK_STREAM)
a_so.bind( ( "localhost", 6666 ) )
a_so.listen(10)

s_so, addr = a_so.accept() a_so.close()

print "Connection from ", addr print
readUntilEOF(s_so)
s_so.sendall("server response line 1\n")
s_so.sendall("server response line 2\n")

s_so.close()
```

Ablauf

- S: `a_so = socket(AF_INET, SOCK_STREAM)`
  - Erzeuge Socket für TCP Verbindung
- S: `a_so.bind( ( ”localhost”, 6666 ) )`
  - Registriere Socket mit Port 6666 und Quell-IP 127.0.0.1
- S: `a_so.listen(10)`
  - Socket wechselt in den Zustand LISTEN
  - 10 Warteplätze für eingehende Verbindungen
- S: `s_so, addr = a_so.accept()`
  - Applikation wartet, bis eine vollständig aufgebaute Verbindung besteht

Ablauf

- C: `c_so = socket(AF_INET, SOCK_STREAM)`
  - Datenstruktur für TCP Verbindung wird angelegt
  - IP Header Version ist 4
  - IP Header Protocol ist 6 (TCP)
- C: `c_so.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)`
  - Socket im Zustand TIME_WAIT kann benutzt werden
- C: `c_so.bind( ( "0.0.0.0", 7777 ) )`
  - IP Header Source IP wird anhand der Routing Tabelle ermittelt
  - TCP Header Source Port ist 7777
  - Socket ist im Zustand CLOSED

Ablauf

- C: `c_so.connect( ( "localhost", 6666 ) )`
  - IP Header Source IP ist 127.0.0.1
  - TCP Header Destination Port ist 6666
  - Client sendet
    IP 127.0.0.1.7777 > 127.0.0.1.6666: S 2476749965:2476749965(0)
  - Socket ist im Zustand SYN_SENT
- S: Empfängt SYN
  - Daten werden in der Listen Queue gespeichert
  - Server antwortet
    IP 127.0.0.1.6666 > 127.0.0.1.7777: S 2976505067:2976505067(0) ack 2476749966
  - Socket ist im Zustand SYN_RCVD

Ablauf

- C: Empfängt SYN-ACK
  - Prüft (Host,Port) Paar und Sequenznummer
  - Client antwortet
    IP 127.0.0.1.7777 > 127.0.0.1.6666: . ack 2976505068
  - Socket ist im Zustand ESTABLISHED
- S: Empfängt ACK
  - Holt Verbindungsdaten aus der Warteschlange
  - Socket ist im Zustand ESTABLISHED
  - `s_so, addr = a_so.accept()` liefert Filedescriptor

Ablauf

- S: `print "Connection from ", addr`
  - Gibt Source IP und Source Port aus
  - Die Daten könnten auch über `s_so.getpeername()` ausgelesen werden.
- S: `print readUntilEOF(s_so)`
  - Liest Daten vom Filedescriptor, bis EOF signalisiert wird

Ablauf

- C: `c_so.sendall("client request line 1\n")`
  - Schreibt 22 Bytes in den Puffer des Sockets
  - Client sendet
    IP 127.0.0.1.7777 > 127.0.0.1.6666: P
    2476749966:2476749988(22) ack 2976505068
- C: `c_so.sendall("client request line 2\n")`
  - Schreibt 22 Bytes in den Puffer des Sockets
  - Client sendet
    IP 127.0.0.1.7777 > 127.0.0.1.6666: P
    2476749988:2476750010(22) ack 2976505068
- C: `c_so.shutdown(SHUT_WR)`
  - Client sendet
    IP 127.0.0.1.7777 > 127.0.0.1.6666: F
    2476750010:2476750010(0) ack 2976505068
  - Socket ist im Zustand FIN_WAIT_1

Ablauf

- S: Empfängt ”client request line 1\n”
  - Bestätigt mit
    IP 127.0.0.1.6666 > 127.0.0.1.7777: . ack 2476749988
  - Applikation wartet weiter auf Daten
- S: Empfängt ”client request line 2\n”
  - Bestätigt mit
    IP 127.0.0.1.6666 > 127.0.0.1.7777: . ack 2476750010
  - Applikation wartet weiter auf Daten
- S: Empfängt FIN
  - Socket ist jetzt im Zustand CLOSE_WAIT
  - Applikaton liest EOF vom Filedeskriptor
  - `s_so.sendall(”server response line 1\n”)`
  - Server sendet
    IP 127.0.0.1.6666 > 127.0.0.1.7777: P
    2976505068:2976505091(23) ack 2476750011

Ablauf

- S: `c_so.sendall(""server response line 2\n")`
  - Schreibt 23 Bytes in den Puffer des Sockets
  - Server sendet
    IP 127.0.0.1.6666 > 127.0.0.1.7777: P
    2976505091:2976505114(23) ack 2476750011
- C: empfängt beide Zeilen
  - Bestätigt mit
    IP 127.0.0.1.7777 > 127.0.0.1.6666: . ack 2976505114
- S: `s_so.close()`
  - Server sendet
    IP 127.0.0.1.6666 > 127.0.0.1.7777: F
    2976505114:2976505114(0) ack 2476750011
- C: Empfängt FIN
  - Client bestätigt:
    IP 127.0.0.1.7777 > 127.0.0.1.6666: . ack 2976505115