Port szkennelés és TCP eltérítés

Juhász Péter Károly "Stone"
<stone@elte.hu>
Modla Ferenc
<modla@inf.elte.hu>

2004. augusztus 13.

Tartalomjegyzék

1. Port szkennelés

Mi is ez? Röviden, egy eljárás, amivel meg lehet állapítani, hogy egy távoli gépen mely portok vannak nyitva, zárva vagy tűzfallal védve. Kicsit hosszabban egy eljárás, ami az összes hálózaton végrehajtott támadás kezdőlépése, akár ha csak Open Relay levelező szerver[1] keresésről van szó, akár egy gép feltöréséről, és milliókat érő vállalati adatok ellopásáról.

1.1. TCP connect() scan

Ez a legalapvetőbb fajtája a TCP vizsgálatnak. Az operációs rendszerünk által nyújtott connect() rendszerhívást használjuk egy kapcsolat megnyitásához a gép minden érdekes portján. Ha a port figyel, a connect() sikerül, különben a port nem elérhető. Egy komoly előnye ennek a technikának, hogy nincs szükség semmilyen különleges kiváltságra. Bármelyik felhasználó a legtöbb UNIX gépen használhatja ezt a hívást. A másik előny a sebesség. Míg egy-egy külön connect() hívás minden

[1] Olyan levelező szerver ami bármilyen feladótól bárminyen címzettnek hajlandó levelet továbbítani. E-mail szmetelők használják.

megcélzott portra egymás után évekig is eltarthat egy lassú kapcsolat esetében, addig sok socketet párhuzamosan használva felgyorsíthatjuk a vizsgálatot. Nem blokkoló I/O használata lehetőséget ad alacsony időtúllépési periódus beállítására, és az összes socketet egyszerre vizsgálhatjuk. A nagy hátulütője az efféle vizsgálatoknak, hogy könnyen detektálhatók és szűrhetők. A célbavett gép logjai egy csokor kapcsolat- és hibaüzenetet mutatnak a kapcsolatot felügyelő szolgáltatásoknak, melyek azonnal lezárják azt.

Íme egy program ami a fentieket illusztrálja (a program használati utasítása elérhető, ha paraméterek nélkül indítjuk):

```
#include <sys/types.h>
#include <sys/select.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <unistd.h>
#include <fcntl.h>
#include <netinet/in.h>

int main(int argc, char* argv[]) {
  int fd[65536];
  struct sockaddr_in sin;
  struct timeval time;
  fd_set fds;
  int i, j, maxfd, ret, optvar;
  int optlen = sizeof(optvar);

  if(argc < 2) {
    printf("Hasznalat: %s <cel ip>\n", argv[0]);
    exit(1);
  }

  printf("%s szkennelese, nyitott portok:\n",argv[1]);
  for(j = 0; j < 256; j++) {
    FD_ZERO(&fds); maxfd = 0;
    for(i = (j * 256); i < ((j + 1) * 256); i++) {
      fd[i] = socket(AF_INET, SOCK_STREAM, 0);
      fcntl(fd[i], F_SETFL, fcntl(fd[i], F_GETFL) | O_NONBLOCK);
      sin.sin_family = AF_INET;
      sin.sin_addr.s_addr = inet_addr(argv[1]);
      sin.sin_port = htons(i);
      memset(sin.sin_zero, 0, sizeof(sin.sin_zero));
      connect(fd[i], (struct sockaddr*)&sin, sizeof(sin));
      FD_SET(fd[i], &fds); if(fd[i] > maxfd) maxfd = fd[i];
    }
    time.tv_sec = 2; time.tv_usec = 0;
    if(select(maxfd + 1, NULL, &fds, NULL, &time) > 0) {
      for(i = (j * 256); i < ((j + 1) * 256); i++) {
        if(FD_ISSET(fd[i],&fds)) {
          getsockopt(fd[i], SOL_SOCKET, SO_ERROR, &optvar, &optlen);
          if(optvar == 0) printf("%6d\n", i);
          FD_CLR(fd[i],&fds);
        }
      }
    }
    for(i = (j * 256); i < ((j + 1) * 256); i++) close(fd[i]);
```

```
  }
  return 0;
}
```

1.2. TCP SYN scan

Ezt a módszert gyakran nevezik "half-open" ellenőrzésnek, mivel nem nyitunk meg egy teljes TCP kapcsolatot. Elküldünk egy SYN csomagot, mintha egy valódi kapcsolatot akarnánk létrehozni, és várunk a válaszra. Egy SYN/ACK jelzi, hogy a port figyel. Egy RST a nem figyelés jele. Ha egy SYN/ACK-t kapunk, azonnal küldünk egy RST-t megbontani a kapcsolatot (valójában a kernel megteszi ezt nekünk). A legfőbb előnye ennek a módszernek, hogy kevesebb helyen loggolják. Sajnos ezeknek a sajátságos SYN csomagoknak a felépítéséhez root jogokra van szükségünk.

És egy program a fentiekre (a program használati utasítása elérhető, ha paraméterek nélkül indítjuk):

```
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <sys/select.h>
#include <stdio.h>

struct {
  unsigned int src;
  unsigned int dst;
  unsigned char dummy;
  unsigned char proto;
  unsigned short len;
  struct tcphdr tcp;
} pseudohdr;

struct {
  struct iphdr ip;
  struct tcphdr tcp;
} packet;

unsigned short cksum(unsigned short *addr, int len) {
  register int nleft = len;
  register unsigned short *w = addr;
  register int sum = 0;
  unsigned short answer = 0;
  while (nleft > 1) {
    sum += *w++;
    nleft -= 2;
  }
  if (nleft == 1) {
    *(u_char *)(&answer) = *(u_char *)w ;
    sum += answer;
  }
  sum = (sum >> 16) + (sum & 0xffff);
  sum += (sum >> 16);
  answer = ~sum;
  return(answer);
}

int main(int argc, char* argv[]) {
```

```
  int i, fd, timeout;
  struct tcphdr tcp;
  struct sockaddr_in sin;
  struct timeval time;
  fd_set fds;

  if(argc < 3) {
    printf("Hasznalat: %s <sajat ip> <cel ip>\n", argv[0]);
    exit(1);
  }
  printf("%s szkennelese, nyitott portok:\n",argv[2]);
  for(i = 1; i < 65537; i++) {
    tcp.source = htons(57043); tcp.dest = htons(i);
    tcp.seq = rand(); tcp.ack_seq = 0;
    tcp.res1 = 0; tcp.doff = 5; tcp.res2 = 0;
    tcp.syn = 1; tcp.fin = 0; tcp.rst = 0;
    tcp.psh = 0; tcp.ack = 0; tcp.urg = 0;
    tcp.window = htons(1024); tcp.urg_ptr = 0;
    tcp.check = 0;
    pseudohdr.src = inet_addr(argv[1]);
    pseudohdr.dst = inet_addr(argv[2]);
    pseudohdr.dummy = 0;
    pseudohdr.proto = IPPROTO_TCP;
    pseudohdr.len = htons(sizeof(struct tcphdr));
    memcpy(&pseudohdr.tcp,&tcp,sizeof(struct tcphdr));
    tcp.check = cksum((unsigned short *)&pseudohdr, sizeof(pseudohdr));
    sin.sin_family = AF_INET;
    sin.sin_port = tcp.dest;
    sin.sin_addr.s_addr = inet_addr(argv[2]);
    fd = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
    sendto(fd, &tcp, 20, 0, (struct sockaddr*)&(sin), sizeof(sin));
    memset(&packet, '\0', sizeof(packet));
    timeout = 0;
    while((packet.tcp.dest != htons(57043)) && !timeout) {
      FD_ZERO(&fds); FD_SET(fd, &fds);
      time.tv_sec = 0; time.tv_usec = 100000;
      if(select(fd + 1, &fds, NULL, NULL, &time)) {
        recvfrom(fd, &packet, sizeof(packet), 0, (struct sockaddr *)&(sin),
                 sizeof(sin));
      } else timeout = 1;
    }
    close(fd);
    if(packet.tcp.syn == 1) printf("%6d\n",i);
  }
  return 0;
}
```

1.3. TCP FIN scan

Vannak olyan helyzetek, ahol a SYN vizsgálat sem elég alattomos. Néhány tűzfal és csomagszűrő figyeli a SYN-eket bizonyos portokon, és bizonyos programok, mint pl. a synlogger vagy a Courtney használható az ilyen vizsgálatok felismeréséhez. A FIN csomagok viszont keresztül juthatnak háborítatlanul. Ezt a módszert Uriel Maimon jellemezte részletesen. Az ötlet pedig a következő: A zárt portoknak a FIN csomagunkra válaszolni kell az alkalmas RST-vel. A nyitott portok-

nak viszont figyelmen kívül kell hagyniuk a kérdéses csomagot. Ez az elvárt TCP viselkedés. Ennek ellenére bizonyos rendszerek (nevezetesen Microsoft gépek) nem megfelelőek ebben a tekintetben. Ők RST-ket küldenek figyelmen kívül hagyva a port állapotát, ezért sérthetetlenek efféle vizsgálatokkal. Más rendszerekkel viszont jól működik. Valójában gyakran hasznos lehet megkülönböztetni egy NT rendszert egy UNIX alapútól, és ezzel a módszerrel ez is elérhető.

Itt a program ami a fentieket szemlélteti (természetesen itt helyes viselkedést feltételezünk):

```
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <sys/select.h>
#include <stdio.h>

struct {
  unsigned int src;
  unsigned int dst;
  unsigned char dummy;
  unsigned char proto;
  unsigned short len;
  struct tcphdr tcp;
} pseudohdr;

struct {
  struct iphdr ip;
  struct tcphdr tcp;
} packet;

unsigned short cksum(unsigned short *addr, int len) {
  register int nleft = len;
  register unsigned short *w = addr;
  register int sum = 0;
  unsigned short answer = 0;
  while (nleft > 1) {
    sum += *w++;
    nleft -= 2;
  }
  if (nleft == 1) {
    *(u_char *)(&answer) = *(u_char *)w ;
    sum += answer;
  }
  sum = (sum >> 16) + (sum & 0xffff);
  sum += (sum >> 16);
  answer = ~sum;
  return(answer);
}


int main(int argc, char* argv[]) {
  int i, fd, timeout;
  struct tcphdr tcp;
  struct sockaddr_in sin;
  struct timeval time;
  fd_set fds;

  if(argc < 3) {
```

```
    printf("Hasznalat: %s <sajat ip> <cel ip>\n", argv[0]);
    exit(1);
  }
  printf("%s szkennelese, nyitott portok:\n",argv[2]);
  for(i = 1; i < 65537; i++) {
    tcp.source = htons(57043); tcp.dest = htons(i);
    tcp.seq = rand(); tcp.ack_seq = 0;
    tcp.res1 = 0; tcp.doff = 5; tcp.res2 = 0;
    tcp.syn = 0; tcp.fin = 1; tcp.rst = 0;
    tcp.psh = 0; tcp.ack = 0; tcp.urg = 0;
    tcp.window = htons(1024); tcp.urg_ptr = 0;
    tcp.check = 0;
    pseudohdr.src = inet_addr(argv[1]);
    pseudohdr.dst = inet_addr(argv[2]);
    pseudohdr.dummy = 0;
    pseudohdr.proto = IPPROTO_TCP;
    pseudohdr.len = htons(sizeof(struct tcphdr));
    memcpy(&pseudohdr.tcp,&tcp,sizeof(struct tcphdr));
    tcp.check = cksum((unsigned short *)&pseudohdr, sizeof(pseudohdr));
    sin.sin_family = AF_INET;
    sin.sin_port = tcp.dest;
    sin.sin_addr.s_addr = inet_addr(argv[2]);
    fd = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
    sendto(fd, &tcp, 20, 0, (struct sockaddr*)&(sin), sizeof(sin));
    memset(&packet, '\0', sizeof(packet));
    timeout = 0;
    while((packet.tcp.dest != htons(57043)) && !timeout) {
      FD_ZERO(&fds); FD_SET(fd, &fds);
      time.tv_sec = 0; time.tv_usec = 100000;
      if(select(fd + 1, &fds, NULL, NULL, &time)) {
        recvfrom(fd, &packet, sizeof(packet), 0, (struct sockaddr *)&(sin),
                 sizeof(sin));
      } else timeout = 1;
    }
    close(fd);
    if(timeout) printf("%6d\n",i);
  }

  return 0;
}
```

1.4. Darabolásos technika

Ez nem egy különálló technika, hanem a többi módszer módosítása. Ahelyett, hogy elküldenénk a próbacsomagot, feldaraboljuk kis IP darabokra. A TCP fejlécet is szétosztjuk több csomagba, hogy a csomagszűrők és társaik nehezebben tudják észrevenni, mit is csinálunk. Legyünk óvatosak ezzel a módszerrel! Néhány programnak problémája lehet ezeknek a pici csomagoknak a kezelésével. Bizonyos snifferek akár az első 36 bytos csomagnál elszállhatnak, pedig még jöhet 24 bytos is. Ugyan ezt a módszert nem fülelik le csomagszűrők és tűzfalak, amik sorba állítanak minden IP darabkát, (mint pl. a Linux CONFIG_IP_ ALWAYS_DEFRAG opcója), sok hálózat nem engedheti meg magának azt a teljesítményrombolást, amit ez okoz. Ezt a lehetőséget ezért csak portscanneléshez használják.

1.5. UDP ICMP "port nem elérhető" scan

Ez a módszer abban (is) különbözik az előzőektől, hogy az UDP protokollt használjuk a TCP helyett. Mivel ez a protokoll egyszerűbb, ezért az ellenőrzése valójában lényegesen bonyolultabb. Ennek az oka az, hogy a nyitott portoknak nem kell nyugtát küldenie válaszul a próbánkra, a lezárt portoknak pedig még hibacsomagot sem kell küldeniük. Szerencsére a legtöbb host küld egy ICMP PORT UNREACH hibát, ha egy csomagot küldünk egy lezárt UDP portra. Ezzel megtudhatjuk, hogy egy port NINCS nyitva, így kizárásos alapon azt is, hogy melyik igen. Mivel nincs garancia arra, hogy akár az UDP csomag akár a hibaüzenet megérkezik, ezért az ilyen UDP ellenőrzőkbe be kell építeni az elveszettnek tűnő csomagok újraküldését (különben lehet egy köteg hamis találatunk). Ugyancsak lassúnak bizonyulhat ez a módszer, ha olyan rendszeren próbálkozunk, mely megfogadta az 1812-es RFC 4.3.2.8 szakaszát, és határt szab az ICMP hibák mértékének. Például a Linux kernel (a net/ipv4/icmp.h-ban) 4 másodpercenként 80-ra korlátozza a cél elérhetetlen üzeneteket, 1/4 mp büntetéssel túllépés esetén. Ezenfelül root jogokra van szükségünk a raw ICMP socket eléréséhez, ami a port elérhetetlen üzenetek olvasásához szükséges.

1.5.1. UDP recvfrom() és write()

A nem root felhasználók ugyan nem tudják közvetlenül olvasni a port elérhetetlen üzeneteket, de a Linux elég ügyes ahhoz, hogy informálja a felhasználót, ha kapott olyat. Például egy második write() hívás egy zárt portra általában sikertelen lesz. Sok szkenner program így ellenőriz. Nem blokkolt UDP socketeken a recvfrom() visszatérő értéke EAGAIN ("Try again", errno 11), ha az ICMP hibát nem kapta meg, és ECONNREFUSED ("Connection refused", errno 111), ha igen. Ez az az eljárás, amivel nem root felhasználók meghatározhatják a nyitott portokat. Rootként is megtehetjük ezt, de nincs sok értelme.

Itt egy példa a fentiekre, ami a második write() visszatérő értékét nézi:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <errno.h>

int main(int argc, char* argv[]) {
  struct sockaddr_in sin;
  int i, fd, ret;
  char out[] = "Stone";

  if(argc < 2) {
    printf("Hasznalat: %s <cel ip>\n", argv[0]);
    exit(1);
  }

  printf("%s szkennelese, nyilt portok:\n",argv[1]);
  for(i = 1; i < 65537; i++) {
    fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = inet_addr(argv[1]);
    sin.sin_port = htons(i);
    memset(sin.sin_zero, 0, sizeof(sin.sin_zero));
    connect(fd, (struct sockaddr*)&sin, sizeof(sin));
    errno = 0;
    ret = write(fd, out, strlen(out));
    usleep(100000);
    ret = write(fd, out, strlen(out));
```

```
    if (ret > 0 && !errno) printf("%6d\n",i);
    close(fd);
  }
  return 0;
}
```

1.6. Idle scan

1.6.1. A techinka

1998 December 17.-én Salvatore Sanfilippo más néven Antirez az `insecure.org` levelezési listájára küldött egy levelet, amiben egy *új szkennelési eljárást* ír le, ami később az *Idle scan* néven épült be a köztudatba.

Az eljárás lehetővé teszi, hogy úgy szkenneljünk egy gépet, hogy egyetlen csomagot sem küldünk neki, mindezt úgynevezett Zombi gépek[2] felhasználásával. A támadás hátterében a IPID (Internet Protocol Identification), azaz az IP csomag azonosítója áll. Ez egy 4 hosszú hexadecimális szám, amit az egyszerűség kedvéért ezentúl decimális alakban használok.

Egy-két alapismeret ami szükséges a további megértéshez:

1. Ahhoz, hogy egy portról eldöntsük, hogy nyitva van-e, kell küldeni egy SYN (synchronize/start) csomagot, amire ő egy SYN ACK-t (synchronize acknowledge) küld vissza, ha nyitva van a port (azaz figyel ott valamilyen szerver) vagy egy RST (reset) csomagot, ha zárva van.

2. Ha egy gép kap egy SYN ACK csomagot aminek nem volt előzménye, akkor küld egy RST-et vissza. A RST csomagot ugyanebben az esetben eldobjuk.

3. Minen IP csomag fejlécében van egy IPID nevü mező, ami már az elöbb említett 4 hosszú hexadecimális számot tartalmazza. A legtöbb operációs rendszer egyszerűen egyesével növeli ezt minden csomagnál.

Tegyük fel, hogy van 3 gép: Támadó, Zombi valamint Célpont. A zombi géppel szemben alapkövetelmény, hogy ne küldjön egyetlen csomagot sem a szkennelés alatt. De ilyen gépeket könnyű találni főleg egyetemeken és nagyobb cégeknél az éjszaka kellős közepén.

Szükségünk van egy-két programra a próba elvégzéséhez. Először is kell egy program, amivel a gépünkre beérkező csomagok IPID-jét figyelhetjük, ehhez használhatjuk az *ethereal*-t, *tcpdump*-ot, vagy akár mi is írhatunk egy egyszerű programocskát. Például a következő kis *Perl* scripttel, ahol a `RawIP` csomag segítségével kapkodjuk el az adatokat:

```
#!/usr/bin/perl

$dev = "lo";

use Net::RawIP qw(:pcap);

$packet = new Net::RawIP;
$s = new Net::RawIP;
$filter = "ip proto \\tcp";
$pcap = $s->pcapinit($dev,$filter,1500,60);
$offset = linkoffset($pcap);
if (fork){ loop $pcap,-1,&check,\@s;}

sub check {
  $packet->bset($_[2],$offset);
```

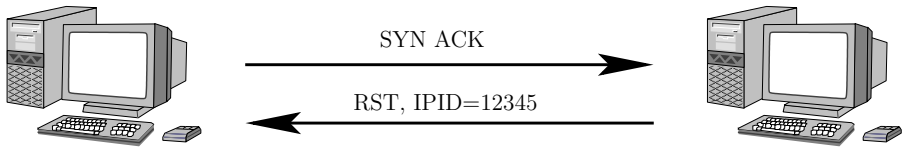
[2]Olyan gépek amiket a támadó arra használ fel, hogy félrevezesse a célpontot és más embereket, akik erről mit sem tudnak, keverjen gyanuba.

```
  ($id,$addr) = $packet->get({ip=>['id','saddr']});
  printf ip2name($addr) . " $id\n";
}

sub ip2name {
  my $addr = shift;
  (gethostbyaddr(pack("N",$addr),AF_INET))[0] || ip2dot($addr);
}

sub ip2dot {
  sprintf("%u.%u.%u.%u", unpack "C4", pack "N1", shift);
}
```

Valamint kell egy másik program amivel olyan csomagokat tudunk küldeni, amilyet akarunk. Ezt is egy *Perl* scripttel oldjuk meg (természetesen a hálózati interfész nevét valamint az IP címeket és portszámokat mindig az aktuális helyzethez kell igazítani):

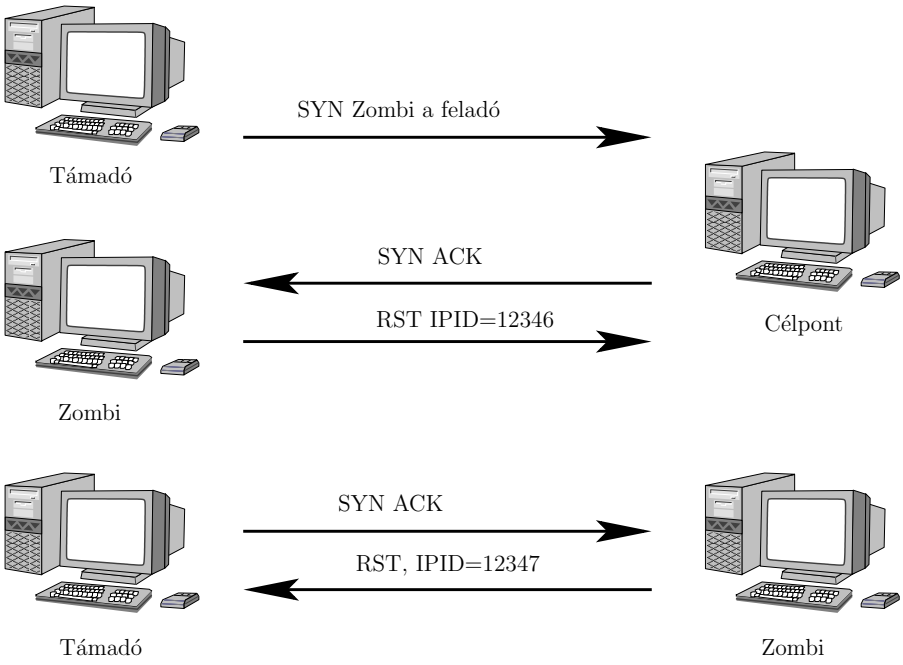
```
#!/usr/bin/perl

use Net::RawIP qw(:pcap);

$dev = "lo";
$daddr = "127.0.0.1";
$dport = 23;
$saddr = "123.234.56.3";
$sport = 2434;

$s = new Net::RawIP;

$s->set({ip => {daddr => $daddr, saddr => $saddr},
         tcp => {dest => $dport, source => $sport, syn => 1, ack => 1}});
$s->send;
```

Most, hogy már mindenünk megvan, kezdődhet a próba! Először is miután kiválasztottuk a zombi gépet, elindítjuk valamelyik sniffer programunkat és küldünk neki egy csomagot a fenti programmal. Ö a csomagra (ami egy SYN ACK csomag volt) egy RST csomaggal válaszol, amiből megtudjuk az aktuális IPID-jét.

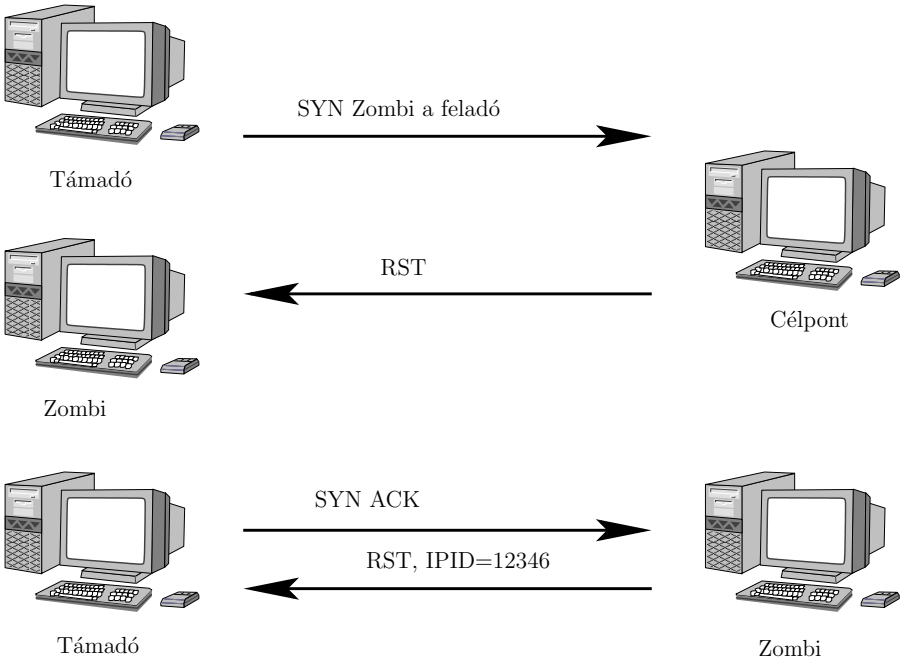


Ezek után a Támadó küld egy SYN csomagot a Zombi nevében a Célpontnak. Majd megint megnézi a már fent eljátszott módszerrel a Zombi gép IPID-jét, ha az eggyel nagyobb mint előzőleg, akkor a port zárva van, ha kettővel, akkor nyitva. Ez azért van, mert ha a port nyitva volt, akkor a Célpont küldött vissza a Zombinak egy SYN ACK csomagot, amire Zombi egy RST-tel válszolt – ezzel növelve az IPID-jét. Ha a port zárva volt, akkor a Célpont RST-et küldött vissza amire a Zombi nem válaszolt így a nekünk küldött két csomag között ő nem küldött más csomagot. Ahogy ezt a következő két ábra is mutatja:

Az első nyitott porttal.



Az második zárt porttal.



A fent szemléltetett eljárás egyik jó tulajdonsága, hogy nem küldünk egy csomagot sem a célpotnak így az esetleg ott lévő IDS (Intruder Detection System) nem minket fog támadóként megnevezni, hanem a zombi gépet. Ezen felül komoly előny, hogy egy másik gép bőrébe bújva (esetleg olyanéba akiben megbízik a célpont) bújva lehet a célt szekennelni. Mivel tegyük fel, hogy a célpontgép az emberünk irodájában van, a tűzfal, csak az ő otthoni gépéről enged át csomagokat, ekkor az otthoni gépet választva zombinak tudjuk az irodai gépet szkennelni.

1.6.2. A védekezés

Több eljárásal is lehet védekezni az ilyen támadások ellen, ezek közül a teljesség igénye nélkül lássunk néhányat:

1. Statefull tűzfal használata – ez olyan tűzfal ami csak olyan csomagokat engednek át amik egy kapcsolathoz tartoznak, azaz egy SYN ACK csomagot nem engednek át, akkor ha a másik irányból nem előzte azt meg egy SYN csomag.

2. Jól konfigurált tűzfalak – ne engedjünk be a külvilág felől benti IP címmel érkező csomagokat a hálózatba.

3. Okos operációs rendszerrel – a GNU/Linux, Solaris vagy az OpenBSD okosabban választja meg az IPID számokat, így nem olyan könnyű figyelemmel kísérni a növekedést. Az OpenBSD véletlenszerűen választ, a Linux 2.4 minden kapcsolódó géphez más sorozatot alkalmaz, valamint 0-t ír az olyan csomagok IPID-jébe ahol a *don't fragment bit* be van állítva.

4. Kimenő hamis csomagok szűrésével – ha nem engedjük ki őket, akkor a hálózatunkból nem lehet ilyen támdást indítani.

2. TCP eltérítés

A következőkben, egy olyan támadást mutatok be, ami egy TCP (Transport Control Protocol) kapcsolat ellen irányul, és a támadót képessé teszi, hogy egy felépített, nem titkosított, mégis sokak által biztonságosnak vélt kapcsolatba adatot csempésszen. A támadást bármelyik gépről meg lehet indítani, ami a két kapcsolati végpont között található, és nem kell hozzá más, mint egy sniffer és egy csomag generátor. Bemutatom a támadást, kitérek arra, hogy hogyan lehet felfedezni, valamint arra, hogy hogyan lehet ellene védekezni.

Az emberek azt hiszik, hogy azzal, hogy egyszer használatos jelszavakat használnak, vagy akár kerberosz ticketeket az azonosításra, azzal megóvják magukat az ellenük elkövetett támadásokkal szemben. Ez részben igaz, mivel a *passzív támadásokat*, amik a jelszavuk lehallgatását és későbbi felhasználását jelenti, kivédik de az *aktív támadásokat*, amik az éppen meglévő kapcsolatuk ellen irányulnak, azokat nem.

2.1. A TCP kapcsolat működése

Ahhoz, hogy megértsük a támadást ismernünk kell, hogy min alapszik, és ez nem más mint a TCP. Röviden ez egy teljesen kétirányú, kapcsolat alapú és megbízható összeköttetés két távoli gép között. Egy kapcsolatot négy adatból lehet azonosítani, ezek: a szerver IP címe, a klines IP címe, a szerver port száma, valamint a kliens port száma. Minden egyes byte adat (itt az IP csomagokra gondolok), ami elküldésre kerül, egyedi azonosítóval rendelkezik, egy 32 bites egész számmal. A kezdő azonosító a kapcsolat építése közben generálódik, szem elött tartva, hogy két csomag ne kapja ugyanazt a számot.

Vezessünk be egy-két jelölést!

SVR.SEQ	a szerver következő csomagjának az azonosítója
SVR.ACK	a következő csomag azonosítója, amit várunk (ez az előző beérkezett csomag azonosítója + 1)
SVR.WIN	a szerver fogadó ablak mérete
CLT.SEQ	a kliens következő csomagjának az azonosítója
CLT.ACK	a következő csomag azonosítója, amit várunk
CLT.WIN	a kliens fogadó ablak mérete

Amikor éppen nincs adat úton, akkor a következő egyenlőségek állnak fenn:

$$\text{SVR.SEQ} = \text{CLT.ACK}, \text{ valamint } \text{SVR.ACK} = \text{CLT.SEQ}.$$

Általánosságban, itt azt is megengedjük, hogy éppen adat legyen úton:

$$\text{CLT.ACK} \leq \text{SRV.SEQ} \leq \text{CLT.ACK} + \text{CLT.WIN}$$
$$\text{SRV.ACK} \leq \text{CLT.SEQ} \leq \text{SRV.ACK} + \text{SRV.WIN}$$

Valamint vezessünk be még egy-két jelölést a csomagok leírásához:

SEG.SEQ	a csomag azonosítója
SEG.ACK	a csomag ACK-ja
SEG.FLG	a csomag flag-jei

Általában a kliens oldalon a SEG.SEQ egyenlő a CLT.SEQ-el valamint a SEG.ACK a SLT.ACK-val.

Most lássuk, hogy hogyan épül fel egy kapcsolat! A kliens az elején ZÁRT állapotban van, a szerver KAPCSOLATRA VÁR-ban. A kliens az első csomagban egy kezdő azonosítót küld (CLT.SEQ_0), valamint egy SYN flaggel jelzi, hogy kalcsolatot akar létesíteni, valamint átvált SYN ELKÜLDVE állapotba. A szerver erre válaszul egy SYN-ACK csomagot küld benne a saját azonosítójával (SRV.SEQ_0), valamint a $\text{CLT.SEQ}_0 + 1$-el mint ACK-val, és átvált SYN MEGKAPVA állapotba. Válaszul a kliens egy ACK csomagot küld a következő értékekkel: $\text{SEG.SEQ} = \text{CLT.SEQ}_0 + 1$, $\text{SEG.ACK} = \text{SRV.SEQ}_0 + 1$, majd átmegy KAPCSOLAT FELÉPÍTVE állapotba. A csomag megérkeztével a szerver is átmegy KAPCSOLAT FELÉPÍTVE állapotba.

Jelenleg a következő értékeink vannak:

$$\text{CTL.SEQ} = \text{CLT.SEQ}_0 + 2 \quad \text{SRV.SEQ} = \text{SRV.SEQ}_0 + 1$$
$$\text{CTL.ACK} = \text{SRV.SEQ}_0 + 1 \quad \text{SRV.ACK} = \text{CLT.SEQ}_0 + 2$$

Amikor KAPCSOLAT FELÉPÍTVE állapotban vannak a gépek, akkor azokat a csomagokat fogadják el, amik a következő zárt intervallumba esnek:

$$[\text{X.ACK}, \text{X.ACK} + \text{X.WIN}], \text{ ahol X} = \text{SRVv.CLI}$$

Ha olyan csomag jön, aminek a sorszáma nincs ebben az intervallumban, akkor a gép egy ACK csomagot küld vissza azokkal az értékekkel amit kapni szeretne (pl. a szerver SRV.SEQ és SRV.ACK értékekkel).

2.1.1. A deszinkronizált állapot

Ez akkor fordul ekő, ha KAPCSOLAT FELÉPÍTVE állapotban vagyunk, nincsen adat úton és

$$\text{SVR.SEQ} \neq \text{CLT.ACK, valamint SVR.ACK} \neq \text{CLT.SEQ.}$$

Ha ilyenkor adat érkezik, két eset fordulhat elő:

1. $\text{SEG.SEQ} > \text{SRV.ACK}$ és $\text{SEG.SEQ} < \text{SRV.ACK} + \text{SRV.WIN}$, ilyenkor a csomagot vagy félretesszük keşőbbi használatra, vagy eldobjuk implementációtól függően, de semmiképpen sem adjuk oda a felhasználónak addig, amíg meg nem jött a hiányzó rész.

2. $\text{SEG.SEQ} < \text{SRV.ACK}$ vagy $\text{SEG.SEQ} > \text{SRV.ACK} + \text{SRV.WIN}$, ilyenkor eldobjuk a csomagot.

2.2. A támadás

A technika abból áll, hogy a támadó deszinkronizált állapotot teremt mindkét végponton (ekkor nem tudnak információt cserélni), és aztán szimulálja nekik az igazi csomagokat.

Amikor sikerült a deszinkronizációt előállítani akkor a következő állapot áll fenn: $\text{SRV.ACK} \neq \text{CLI.SEQ}$, valamint $\text{SRV.SEQ} \neq \text{CLI.ACK}$. Most nézzük, hogy mi történik ha például küld a kliens egy csomagot a szervernek (a másik irány hasonlóan néz ki)!

A kliens elküldi a csomagot CLI.SEQ és CLI.ACK értékekkel, mivel az gondolja ezt várja a szerver. De mivel a szerver nem erre számít eldobja a csomagot. De a támadó látja a próbálkozást és egy pont ugyanolyan csomagot küld, csak kicseréli a SEG.SEQ-et SVR.ACK-ra, a SEG.ACK-t SRV.SEQ-re, valamint az ellenörző összeget is újraszámítja, hogy a csomag sértetlennek nézzen ki. Ezt a szerver már örömmel elfogadja.

Vezessük be a következő jelöléseket:

$$\text{CLT.2.SRV.OFFSET} = \text{SRV.ACK - CLI.SEQ}$$
$$\text{SRV.2.CLT.OFFSET} = \text{CLI.ACK - SRV.SEQ}$$

Ekkor a támadónak a következő képletek szerint kell a klinestől a szerver felé menő csomagokat módosítani:

$$\text{SEG.SEQ} := \text{SEG.SEQ} + \text{CLT.2.SRV.OFFSET}$$
$$\text{SEG.ACK} := \text{SEG.ACK} - \text{SRV.2.CLT.OFFSET}.$$

Ha a támadó minden csomagot el tud kapni mind a szervertől a kliens felé, mind vissza, akkor le tudja szimulálni őket egymásnak úgy, hogy ezt nem veszik észre, sőt bármilyen adatot tud mind betenni a kapcsolatba, mind kivenni. Vegyünk egy példát, ami remélem már sehol a világon nem fordulhat elő, de most legyen: a rendszergazda telnettel jelentkezett be a gépére távolról. A támadó sikeresen szimulálja a kapcsolatot. A klienstöl a szerver felé irányuló kapcsolatba egy új csomagot csempészik bele, legyen ez a következő: `echo "ghost:x:0:0:Ghost:/:/bin/sh" > /etc/passwd; echo "ghost::0:0:Ghost:/:/bin/sh" > /etc/shadow`. Valamint az esetleges fölösleges válaszokat kiszűri. Most ebből az egészből a rendszergazda nem vett semmit észre, esetleg akkor, mikor a támadó egyszer csak kidobta a saját gépeéről, és a következő próbálkozásnál, mikor be szeretne lépni, már nem jó a jelszava.

2.2.1. Az ACK vihar

A támadás szépséghibája, hogy sok ACK csomagot generál. Ha a gép kap egy csoamgot amit nem fogad el, akkor küld egy ACK csomagot azokkal az értékekkel, amiket kapni szeretne. Ez a csomag is elfogadhatatlan a másik oldal számára (mivel mindketten deszinkronizált állapotban vannak), így ő is küld egy ACK csomagot, és máris egy végtelen ciklusban vagyunk.

Mivel ezek a csomagok nem szállítanak adatot, ha elvesznek nem küldik őket újra, és mivel a hálózat nem tökéletes, ezért csomagok néha elvesznek. Ha itt a viharból elveszik akár csak egyetlen csomag is, akkor a vihar elül. Érdekesség, hogy ezek a viharok önszabályozók: minél több a vihar, annál nagyobb a hálózati kihasználtság, annál nagyobb a csomag veszteség, és így annál több vihar hal el.

Minden egyes alkalommal vihar keletkezik, amikor a szerver vagy a kliens adatot küld. Nem keletkezik vihar, ha a gépek gyüjtögetik az olyan csomagokat amik nem jók nekik, de majd valamikor jók lesznek. (Ez az első eset amit a 2.1.1-es szakaszban leírtam.) De ilyenkor ezek az eltárolt csomagok galibát okozhatnak a későbbiekben.

A vihar ellen lehetséges védekezés az, ha a támadó nem endegi át az *igazi* csomagokat.

2.2.2. A kapcsolat felépítése

Most két techinka következik arra, hogy a deszinkronizált állapotot előidézzük. Más techinkák is lehetnek, de azokat már az olvasóra bízzuk.

A korai deszinkronizáció. Ez a technika a kapcsolat felépítését célozza meg. A támadó a szervertől a kliens felé menő SYN-ACK csomagokra figyel. Amint jön egy ilyen, rögtön küld a szervernek egy RST csomagot és egy SYN csomagot ugyanarra a portra, de egy mások sorszámmal (ATK.SEQ_0). Ekkor a szerver zárja a kapcsolatot, és nyit egy újat egy új sorszámmal ($\text{SVR.SEQ}'_0$). És elküldi a SYN-ACK csomagot a kliensnek (aki nem más mint a támadó). Ekkor a támdó küld egy ACK csomagot a szervernek, mire az KAPCSOLAT FELÉPÍTE állapotba megy át. A klines már akkor állapotot váltott amikor az első SYN-ACK csomag megjött a szervertől.

Most már mindketten deszinkronizált állapotban vannak. Valamint tudjuk a következőket:

$$\text{SVR.2.CLT.OFFSET} = \text{SVR.SEQ}'_0 \text{ - SVR.SEQ}_0$$
$$\text{CLT.2.SVR.OFFSET} = \text{ATK.SEQ}_0 \text{ - CLT.SEQ}_0$$

A technika sebezhetősége a CLT.2.SVR.OFFSET-en múlik, mert rossz érték esetén esetleg a klines csomagjait a szerver elfogadja, ha azok ablakméreten belül érkeznek és ez még kellemetlenségekhez vezethet.

Az üres adat deszinkronizáció. Ez a technika olyan adatot küldésén alapul, amik nem befolyásoljál a kliens és a szerver működését. Vegyünk például egy telnet kapcsolatot! A telnet kapcsolatban az IAC NOP (No operation) utasítasra a telnet démon nem csinál semmit, ezért elég sok ilyen csomag elküldésével deszinkronizációt idézhetünk elő. Ez a kliens irányába is működik.

Olyan kapcsolattal, ami nem tud olyan adatot szállítani, ami nem befolyásolja láthatóan mind a szerver, mind a kliens működését, ez a támadás alkalmazhatatlan.

2.3. Védekezés

Ez a támadás többféleképpen is detektálható. A továbbiakban eseket a módszereket mutatom be.

Deszinkronizáció detektálaśa Nyilván ha összetudjuk haszónlítani a végpontokon lévő ACK és SEQ számokat könnyen lebuktatható a támadás. De ez csak akkor kivitelezhető ha úgy tudjuk a kapcsolaton átküldeni a számokat, hogy a támadó ne változtassa meg azokat is. Vagy esetleg egy teljesen maś úton juttatjuk el azokat az egyik hejről a másikra.

Az ACK vihat detektálása Mivel a támadás sok ACK csomag küldözgetésével jár ez is nagyon árulkodó jel lehet. Ha ezt nem is figyeljük snifferrel, akkor is észrevehetjük abból, hogy miközben mi alig komunikálunk a hálózat kihasználtsaǵa hirtelen megugrik, annyira, hogy a többi kapcsolatunk is lelassul és akadozik.

Sikertelen kapcsolatok A támdás azon verzióját ami a kapcsolatot a felépítésnél támdja meg onnan lehet néha észlelni, hogy sikertelen kapcsolatokat produkálhat. Mivel ha a támdó csomagjai elvesznek a hálózaton, akkor félig kiépített vagy deszinkronizált kapcsolatok jönnek létre, amik adatszállításra alkalmatlanok.

2.4. Példák

Irodalomjegyzék

[1] Idle Scanning and related IPID games, http://www.insecure.org/

[2] Salvatore "Antirez" Sanfilippo - Dumbscan, http://www.kyuzz.org/antirez/papers/dumbscan.html

[3] Laurent Joncheray - Simple Active Attack Against TCP, http://www.insecure.org/stf/iphijack.txt

[4] Transmission Control Protocol - RFC793 - http://www.rfc-editor.org/rfc/rfc793.txt

[5] Internet Protocol - RFC791 - http://www.rfc-editor.org/rfc/rfc791.txt