© Kiskapu Kft. Minden jog fenntartva

A Linux csomagszűrő felépítése (2. rész)

Gianluca elmeséli, hogy a rendszermagon áthaladva a TCP-feldolgozó végül hogyan fogja el a csomagokat.

Sorozatunk előző részben azt tárgyaltuk, hogyan jut el egy csomag a fizikai szintről – a hálózati kábelről – a felsőbb szinteken található hálózati verembe. A csomagot egészen a 3. szintig kísértük el, ahol az IP-nek már nincs több dolga, és a feldolgozást a TCP vagy az UDP veszi át. Ebben a cikkben utazásunkat a 4. szint elemzésével, a `PF_PACKET`-protokoll felépítésével és a foglalati szűrő kapcsolataival fejezzük be.

Eltekintve az IGMP- és ICMP-feldolgozástól, amelyeket a rendszermag dolgoz fel, a csomag útja az alkalmazás felé kétféleképpen végződhet: vagy a `tcp_v4_rcv()`, vagy a `udp_rcv()` függvénynek adódik át. A TCP feldolgozása egy kicsit nyakatekert, mivel a protokoll maga egy FSM (Final State Machine, vagyis véges állapotú gép), és egy egész sor közbenső állapota lehetséges. Gondoljunk csak a TCP-foglalat különböző állapotaira: `listening` (kapcsolatra vár), `established` (a kapcsolat létrejött), `closed` (a kapcsolat véget ért), `waiting` (a kapcsolat lezárása folyamatban) stb.

Hogy leegyszerűsítsük a leírást, a folyamatot a következő lépésekre bontjuk fel:

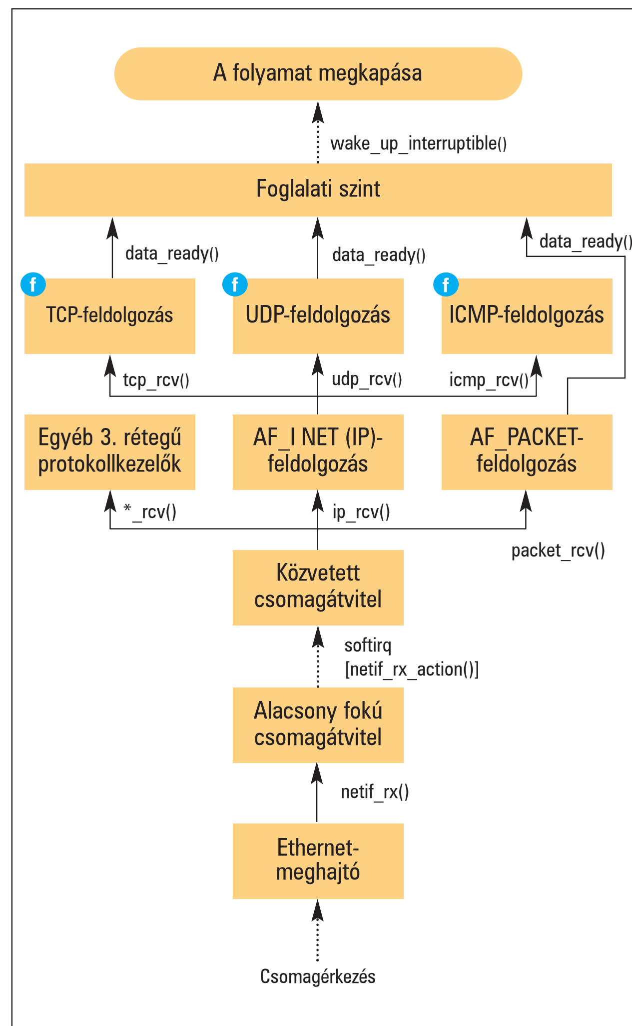
- A `tcp_v4_rcv()` függvény (***net/ipv4/tcp_ipv4.c***) ellenőrzi a TCP-fejléc sértetlenségét.
- A `__tcp_v4_lookup()` függvény felkutatja azt a foglalatot, amely erre a csomagra vár.
- Ha nem találja meg, megteszi az ilyenkor szükséges lépéseket (többek között utasítja az IP-t, hogy egy ICMP-hibaüzenetet hozzon létre).
- Ha megvan a foglalat, a csomagot továbbadja a `tcp_v4_do_rcv()`-nek, mely kézbesíti azt (`sk_buff`) és a hozzá tartozó foglalati kiépítést.

Mindemellett ez a függvény felelős minden egyéb teendő elvégzéséért, mely a csomag állapotától függően szükséges.

A csomagszűrő szempontjából a TCP-feldolgozás során meghívott legutolsó függvény az érdekes, amelyről szó is esett. Az `sk_filter()` függvény a `tcp_v4_do_rcv()` hívás legelején kerül meghívásra, ami – mint majd látni fogjuk – minden csomagszűrővel kapcsolatos varázslatért felelős. Honnan tudja a rendszermag, hogy egy adott foglalatra érkező csomag esetén meg kell hívnia a csomagszűrőt? Ez az adat a foglalathoz tartozó foglalati felépítésben van tárolva. Ha a `setsockopt()` rendszerhívással valamilyen szűrőt kapcsolunk, a felépítésben a megfelelő érték valamilyen igaz értéket vesz fel, és a TCP-kezelőhívás tudni fogja, hogy meg kell hívnia az `sk_filter()`-t.

Azok számára, akik már járatosak a foglalatok programozásában és fel tudják idézni, hogy kapcsolat érkezésekor a kiszolgálóalkalmazásból egy új példány `fork()`-olódik, megemlítjük, hogy a szűrőt először egy kapcsolatra váró (`listening`) foglalatra csatlakoztatjuk. Ezután új kapcsolat érkezésekor a szűrő lemásolódik az újonnan létrejött foglalatba. Amennyiben érdekelnek a részletek, vess egy pillantást a `tcp_create_openreq_child()` függvényre a ***net/ipv4/tcp_minisocks.c*** fájlban.

A csomagkezelőhöz visszatérve: a csomag további sorsa a szűrő kimenetétől függ; ha a csomag megfelel a szűrőben található feltételnek, akkor a feldolgozás tovább folyik, máskülönben a



csomag eldobódik. Ezen túlmenően a csomagszűrő meghatározhatja azt a legnagyobb csomagméretet, amelyet feldolgozásra továbbít (a csomagok kilógó részét az `skb_trim()` függvény vágja le). A foglalat állapotától függően a csomag útja több irányban folytatódhat. Ha a kapcsolat már létrejött, a csomagot a `tcp_rcv_established()` függvény kapja meg. Ennek a függvénynek nagyon fontos feladata van: elvégzi a bonyolult TCP-visszaigazolási rendszer kezelését és a fejlécek feldolgozását, amelyekre most nem térünk ki. Egyedül a foglalathoz (`sk`) tartozó `data_ready()` függvényt említjük meg,

A memória címeinek szűrése

Család	*Foglalattípus*	*Protokollnév*	*A memória ezzel kezdi*
PF_PACKET	STOCK_RAW	Nyers csomag	Összekötőréteg-fejléc
PF_PACKET	STOCK_DGRAM	Csomag	IP-fejléc
PF_PACKET	STOCK_RAW	Nyers IP	IP-fejléc
PF_PACKET	STOCK_DGRAM	UDP	UDP-fejléc
PF_PACKET	STOCK_DGRAM	TCP	TCP-fejléc

mely meghívja a `sock_def_readable()`-t, ez a `wake_up_interruptible()` függvény segítségével felébreszti a fogadó alkalmazást.
Szerencsére az UDP-feldolgozás ennél jóval egyszerűbb. Az `udp_rcv()` függvény, miután néhány ellenőrzést elvégzett a csomagon, az `udp_v4_lookup()` függvényt hívja meg, ami megkeresi a fogadó foglalatot, és meghívja az `udp_queue_rcv_skb()`-t. Ha nem talál megfelelő foglalatot, a csomagot eldobja.
Az utóbbi függvény meghívja a `sock_queue_rcv_skb()`-t (***sock.h***), amely a csomagot a foglalat várakozó sorába helyezi. Ha ebben az átmeneti tárban már nincs több hely, a csomag eldobódik. A szűrést is ez a függvény végzi a TCP-feldolgozásnál látottakhoz hasonlóan, vagyis az `sk_filter()` segítségével. Végül pedig a `data_ready()` függvény hívódik meg, és ezzel az UDP-csomag feldolgozása véget is ért.

Mi történik a PF_PACKET-csomagokkal?

A `PF_PACKET` család külön említést érdemel. Ebben az esetben a beérkező csomagok mindenféle feldolgozás nélkül az alkalmazás foglalatába kerülnek. Ismerve a csomagfeldolgozó rendszert, amelyet az eddigiekben tárgyaltunk, megállapíthatjuk, hogy nem túl bonyolult dologról van szó.
Ha létrehozunk egy `PF_PACKET` foglalatot (lásd a `packet_create()` függvényt a ***net/packet/af_packet.c*** fájlban), a `NET_RX softirq` által használt listához egy új protokoll adódik hozzá. Az ehhez a protokollcsaládhoz tartozó csomagtípus bekerül az általános (`ptype_all`) vagy a protokolljellemző listába (`ptype_base`), fogadófüggvénye pedig a `packet_rcv()` lesz. Bizonyos okok miatt, amelyeket csak később tisztázunk, az újonnan létrehozott foglalatok címe a csomag típusának megfelelő adatszerkezetben tárolódik. Ez a cím logikailag valójában a rendszermagnak nem ehhez a részéhez tartozna, csak később, egy 4. szintbeli kódban lesz rá szükség, ahol a foglalati adatok feldolgozásra kerülnek. Ezért ebben az esetben ez az adat a bejegyzett protokoll adatmezői között hozzáférhetetlen adatként tárolódik – egy lefoglalt mező a szerkezetben, mely a protokoll működéséhez szükséges.
Ettől a pillanattól kezdve az összes gépre belépő csomag, mely túljut a szokásos fogadási eljáráson, a `net_rx_action()` futása során átadódik a `PF_PACKET` fogadófüggvényének. Ennek a függvénynek első dolga, hogy megpróbálja visszaállítani a kapcsolati szintű fejlécet, ha a foglalat típusa `SOCK_RAW` (emlékezzünk vissza a „Csomagszűrő: bájtok leszippantása a hálózatról” című írásomra a Linuxvilág 2001. június–júliusi számában, miszerint a `SOCK_DGRAM`-foglalatok nem látják a kapcsolati réteg fejlécét).
Ezt a fejlécet vagy a hálózati kártya távolítja el (vagy bármilyen hálózati eszköz, amely a csomagot elkapja), vagy a hálózati kártya eszközmeghajtója (megszakításkezelője). Hálózati kártyák esetén szinte kivétel nélkül ez utóbbi a helyzet. A kapcsolati szintű fejléc egyáltalán nem állítható vissza amennyiben a hálózati eszköz távolította el, mivel ebben az esetben az adat soha nem kerül a rendszermemóriába, és a hálózati eszközön kívül teljesen láthatatlan. A fejlécvisszaállító művelet egyáltalán nem költséges, köszönhetően a rendszermag által nyilvántartott `skbuff` átmeneti táraknak. A következő lépéssel ellenőrizhető, hogy a fogadófoglalatra van-e szűrő kapcsolva. Ez a rész egy kicsit macerás, lévén a szűrővel kapcsolatos adatok a foglalati felépítésben tárolódnak, amelyet még nem ismerünk, mivel a protokollverem legalján vagyunk. `PF_PACKET`-foglalatok esetén azonban – melyeknek kötelezően meg kell kerülniük ezt a vermet – tudniuk kell a foglalati felépítés címét. Ez megmagyarázza, hogy a foglalat létrehozása idején miért kellett a foglalati felépítés címét a protokollblokk saját részébe beírnunk; ez viszonylag tiszta módot biztosít arra, hogy a csomag fogadásakor hozzáférjünk az adathoz.
Azáltal, hogy a foglalati felépítés kéznél van, a rendszermag képes eldönteni, hogy létezik-e szűrő a foglalathoz, és hogy meg kell-e azt hívnia (az `sk_run_filter()` híváson keresztül). Mint általában, a csomag sorsáról a szűrő dönt, vagyis hogy szükség van-e rá, esetleg el kell-e dobni (`kfree_skb()`), vagy megfelelő méretűre kell-e vágni (`pskb_trim()`).
Ha a csomagot elfogadjuk, a következő lépésben az azt tartalmazó `sk_buff`-ról másolatot kell készítenünk. Erre a másolatra feltétlenül szükség van, mivel a `PF_PACKET` is felhasznál egy `sk_buff`-ot, és az esetlegesen később következő szabályszerű protokollok is. Képzeljük el azt az esetet, amikor valamely program megnyit egy `PF_PACKET`-foglalatot, miközben ugyanabban az időben egy másik program a Weben böngészik. Minden egyes csomag esetén, mely a webkapcsolathoz tartozik, a `net_rx_action()` még azelőtt meghívja a `PF_PACKET` feldolgozó eljárásait, mielőtt azok az alapértelmezett IP-protokollok feldolgozó eljárásaihoz kerülnének. Ebben az esetben a csomagból két másolat szükséges: egy a szabályszerű fogadófoglalat számára, mely a böngészőnek továbbítódik, egy pedig a `PF_PACKET`-nek, amelyet az elkapó (sniff) program kap meg. Fontos, hogy a csomagról csak azután készül másolat, miután átjutott a szűrőn. Ily módon csak azok a csomagok igényelnek további CPU-időt, amelyek a szűrő feltételeinek ténylegesen megfelelnek. Ha a csomagszűrés alkalmazási szinten valósulna meg (vagyis ha a `PF_PACKET`-et foglalati szűrő nélkül használnánk), akkor a rendszermagnak az összes beérkező csomagról másolatot kellene készítenie, ami a teljesítmény szempontjából nézve meglehetősen rossz lenne. Szerencsére a csomagmásolás mindössze annyit jelent, hogy az `sk_buff` mezőit kell lemásolnunk, és nem magát a csomagban lévő adatokat (amelyre mind az `sk_buff`-ban, mind a másolatban ugyanaz a mutató hivatkozik).
A `PF_PACKET` feladata a fogadófoglalaton a `data_ready()` függvény meghívásával ér véget, csakúgy, mint a TCP és UDP esetén is. Ezen a ponton a `recv()` vagy `recvfrom()` függvényekre várakozó program feléled, és a csomag kézbesítése a végéhez ér.

Alvó feladatok

Elgondolkoztál már azon, hogyan kerül egy feladat alvó állapotba, ha egy adott foglalaton meghívja a `recv()`, a `recvfrom()` vagy a `recvmesg()` rendszerhívások egyikét?

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva

A folyamat valójában nagyon egyszerű: a rendszermagon belül minden `recv` függvényt a többé-kevésbé közvetlenül meghívott `sock_recv()` függvény valósít meg (***net/socket.c***). Ez a függvény pedig meghívja a `recvmsg()` függvényt, mely a foglalati felépítés protokolljellemző műveletei között van bejegyezve. Például a `PF_PACKET` protokoll esetében ez a függvény a `packet_recvmsg()`. Ez a protokollfüggő `recvmsg()` függvény többek között előbb vagy utóbb meghívja az `skb_recv_datagram()` függvényt, mely minden protokoll esetén az általános adatgramfogadás kezeléséért felelős függvény. Ez az utóbbi függvény a `wait_for_packet()` (***net/core/datagram.c***) meghívásával blokkolja az adott programot, mely ezután `TASK_INTERRUPTIBLE` állapotba, vagyis alvó állapotba lép, és egyúttal azt a foglalat várakozási sorába helyezi. A program addig pihen ott, míg egy új csomag érkezése esetén a `wake_up_interruptible()` függvény meg nem hívódik, mint azt az előző bekezdésekben is láthattuk.

Mire jó maga a szűrő?

A szűrő megvalósításának fő része a ***core/filter.c*** fájlban található, míg a `SO_ATTACH/DETACH_FILTER` `ioctl`-ek a ***net/core/sock.c*** fájlban kerülnek feldolgozásra. Kezdetben a szűrő az `sk_attach_filter()` függvény meghívásával kapcsolódik a foglalatra, amely – miután az épségét ellenőrizte (`sk_chk_filter()`) – a felhasználói szintről a rendszermag állapotterébe másolja át. Ez az ellenőrzés gondoskodik arról, hogy a szűrő parancsfeldolgozójába nehogy oda nem illő kód kerüljön. Végül a szűrő báziscíme átmásolódik a foglalati felépítés szűrőmezőjébe, ahonnan majd később meghívódik.
A csomagszűrő támogatás az `sk_run_filter()` függvényben valósul meg, mely egy `skb`-t (pillanatnyi csomagot) és egy szűrőprogramot kap. Ez utóbbi csak egy egyszerű BPF-utasításokból álló tömb, mely numerikus vezérlők és operandusok sorozatát takarja. Az `sk_run_filter()` csak egy egyszerű BPF-parancsértelmezőt valósít meg (egy virtuális CPU-t, ha úgy tetszik), teszi mindezt felettébb logikus módon; egy hosszú `switch/case` kifejezés határoz a vezérlők alapján, és végzi el a szükséges műveleteket az emulált regisztereken és a memóriában. Az emulált memóriaterület, ahol a kód lefut, természetesen a csomag adott szintnek megfelelő környezetében található (`sk->data`). A szűrő kódja egészen addig végrehajtódik, míg el nem jut egy `BPF RET` utasításig, ezután a függvény kilép.
Vedd észre, hogy az `sk_run_filter()` közvetlenül a `PF_PACKET` feldolgozó eljárásaiból hívódik meg! A foglalati szintű fogadóeljárások (TCP, UDP és nyers IP) az `sk_filter()` (***sock.h***) behúzófüggvényen haladnak keresztül, mely amellett, hogy belsőleg meghívja az `sk_run_filter()`-t, a csomagokat is a megfelelő méretűre vágja.

Kapcsolatok a csomagszűrőhöz

Utunk a rendszermag csomagszűrő függvényei között a végéhez érkezett. Érdekes levonni néhány következtetést a csomagszűrő belépési pontjaival kapcsolatban. Mint láttuk, a rendszermagon belül három jól elkülöníthető pont van, ahol a szűrő meghívódhat: a TCP és UDP fogadófüggvények (4. szint), és a `PF_PACKET` fogadófüggvénye (kettő és feledik szint). A nyers IP-csomagok kiszűrődnek, mivel ugyanazt az utat teszik meg, mint az UDP-csomagok (nevezetesen: `sock_queue_rcv_skb()`, mely adatgramok fogadására szakosodott).
Fontos megjegyezni, hogy a szűrő minden szinten az adott szintnek megfelelő környezetben fut le. Azaz ahogy egyre feljebb halad a veremben, egyre kevesebb adat áll rendelkezésére. A `PF_PACKET`-foglalatok esetén a szűrő a 2. szintnek megfelelő adatokkal rendelkezik, mely a `SOCK_RAW`-foglalatoknak a teljes kapcsolati szintű adatkereteket magában foglalja, vagy pedig 4. szintnek megfelelően a teljes IP-csomagot TCP- és UDP-foglalatoknak (alapvetően a kapuk számát, és még néhány egyéb hasznos adatot tartalmaz). Ebből következően a 4. szintű csomagszűrő értelmetlen. Természetesen az alkalmazás adatterülete mindig a szűrő rendelkezésére áll, annak ellenére is, hogy csak nagyon ritkán van rá szükség, vagy éppen semmikor. A 4. szint használhatatlanságára látható tökéletes példa az 1. és 2. listákon (➲ http://www.linuxvilag.hu/Csomagszuro webhelyen), melyeken egy egyszerű UDP-kiszolgálót figyelhetünk meg. A szűrő csak azokat a csomagokat fogadja, amelyek az ***lj*** karaktersorozattal kezdődnek (vagyis `0x6c6a` hexadecimálisan). Ahhoz, hogy ki tudjuk próbálni, fordítsuk le a programot, és nevezzük ***udprcv***-nek, majd futtassuk le. Utána fordítsuk le a 2. programot, és nevezzük ***udpsnd***-nek, végül futtassuk le azt is:

```
# ./udpsnd 127.0.0.1 "hello world"
```

Ennek hatására az ***udprcv*** semmit sem jelenít meg. Most próbáljuk meg a következő ***lj***-vel kezdődő karaktersorozattal:

```
# ./udpsnd 127.0.0.1 "lj rules"
```

Ezúttal a karaktersorozat annak rendje és módja szerint megjelent, mivel a csomag tartalma megfelel a szűrőnek. Egy másik fontos dolog, mellyel a szűrőkódolónak tisztában kell lennie, az, hogy minden egyes foglalattípus esetén (`PF_PACKET`, nyers IP, vagy TCP/UDP) más-más szűrőt kell használnia.
A szűrő a memóriahivatkozásokkor valójában relatív címzési módot alkalmaz, mely minden esetben a csomagban található adat első bájtjához igazodik. A leggyakrabban alkalmazott protokollcsaládokhoz a szűrő memóriacímeit *táblázatunk* tartalmazza.
A Linux Journal 2001. júniusi számában található cikkben a `tcpdump -dd` paranccsal nyert szűrőkód csak `PF_PACKET`-csomagokkal érvényes, mivel a 2. szintnek megfelelő szűrőkódot állít elő (ugyanis feltételezi, hogy a `0`-s cím a kapcsolati szintű keret kezdete).

Összegzés

Érdekes követni egy csomag kalandos útját a rendszermagon keresztül. Utunk során találkoztuk alapvető rendszermag-adatszerkezetekkel (`skbuff`-ok), jellegzetes programozási módszereket fedeztünk fel (felépítések használata függvényekre mutató mutatókkal a C++ objektumok hatékony választási lehetőségeként), és megismertünk néhány új 2.4-es rendszermagbeli rendszert (`softirq`-k).
Ha többet szeretnél tudni erről a témáról, fegyverkezz fel a rendszermag forrásával, keress egy jó szövegszerkesztőt, hörpints fel egy pohár kávét, majd kezdj kutakodni a kódban. A részvétel ingyenes, a szórakozás garantált!

Linux Journal március, 95. szám

Gianluca Insolvibile
a 0.99pl4-es rendszermag óta Linux-rajongó. Jelenleg hálózati és digitális videokutatással és -fejlesztéssel foglalkozik. Elérhető a g.insolvibile@cpr.it címen.