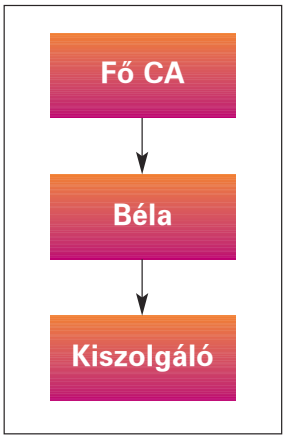
Bevezetés az OpenSSL programozásába

Égető szükséged van egy egyszerű webkiszolgáló-ügyfél párra? Most megtudhatod, miért neked való az OpenSSL.

Egy TCP-n alapuló hálózati alkalmazást legegyszerűbben és leggyorsabban az SSL használatával tehetünk biztonságossá. Ha C nyelven dolgozol, talán a legjobb választás az OpenSSL (➲ http://www.openssl.org). Az OpenSSL ***Eric Young*** SSL/TLS-alapú SSLeay csomagjának szabad forrású változata, amely BSD stílusú felhasználói szerződés hatálya alá esik. Az OpenSSL-hez tartozó leírás és a példaprogramok sajnos sok kívánnivalót hagynak maguk után. A súgóoldalak – ahol vannak – meglehetősen jók, de a mélyebb összefüggéseket gyakran figyelmen kívül hagyják, mivel ezek elsősorban nem tankönyvnek, hanem vonatkozó kézikönyvnek íródtak. Az OpenSSL API hatalmas és bonyolult, ebben a cikkben nem is kíséreljük meg a maga teljességében bemutatni. A célunk inkább az, hogy a súgóoldalak alapján megtanítsunk hatékonyan dolgozni. Ebben a cikkben, amely egy kétrészes sorozat első fele, felépítünk egy egyszerű webes ügyfelet és kiszolgálót, amelyek az OpenSSL alapvető tulajdonságait használják ki. A második cikkben a fejlettebb sajátosságokat mutatjuk majd be, többek között a kapcsolat újrafelvételét és az ügyfél azonosítását.



Kiterjedt tanúsítványlánc

Felteszem, tisztában vagy az SSL és a HTTP főbb fogalmaival. Ha nem, az ismerkedést érdemes az RFC-kkel kezdeni (lásd a ***Kapcsolódó címeket***).

Terjedelmi okok miatt a forráskódokból csak szemelvényeket közlünk. A teljes forráskód a szerző weblapjáról tölthető le (➲ http://www.rtfm.com/openssl-examples).

Programok

Ügyfélprogramunk egyszerű HTTPS-ügyfél (lásd RFC 2818). SSL-kapcsolatot kezdeményez a kiszolgálóhoz, és a kapcsolaton keresztül HTTP-kérést küld át. Ezután várja a kiszolgáló válaszát, majd azt kiírja a képernyőre. Ez a fetch- és a cURL-programok tevékenységének nagymértékben leegyszerűsített változata.

A kiszolgálóprogram egyszerű HTTPS-kiszolgáló, amely az ügyfelek által kezdeményezett TCP-kapcsolatot várja. Amikor egy kérés beérkezik, SSL-kapcsolatot épít fel. Felépülte után beolvassa az ügyfél HTTP-kérését, majd HTTP-választ küld neki, ezt követően pedig lebontja a kapcsolatot.

Első feladatunk a környezeti objektum beállítása (`SSL_CTX`). Ez a környezeti objektum használható a későbbiekben új kapcsolati objektumok létrehozására minden egyes SSL-kapcsolathoz. A kapcsolati objektumok pedig az SSL kézfogási, olvasási és írási műveleteihez nyújtanak segítséget.

Ennek a megközelítésnek két előnye van. Először is a környezeti objektum használatával sok szerkezetnek csak egyszer kell kezdőértéket adnunk, ami teljesítménynövelő hatású. A legtöbb alkalmazásban minden SSL-kapcsolat ugyanazokat a kulcsadatokat, tanúsítványhitelesítő (CA) listákat stb. fogja használni. Ahelyett, hogy ezeket az adatokat minden kapcsolatra újra betöltenénk, a program indulásakor egyszerűen a környezeti objektumba töltjük be őket. Amikor új kapcsolatot szeretnénk létrehozni, csak a környezeti objektumra kell hivatkoznunk.

A környezeti objektum használatának második előnye, hogy több SSL-kapcsolat között teszi lehetővé az adatok megosztását, úgy, mint az SSL-kapcsolatgyorstár a kapcsolat folytatásához. A környezet előkészítése négy elsődleges feladatból áll, mindegyiket az `initialize_ctx()` függvény végzi el az *1. listán* (32. oldal) látható módon.

Mielőtt az OpenSSL-t bármire is használhatnánk, a programkönyvtárat elő kell készíteni. Ezt a feladatot végzi el az SSL_library_init(), amely betölti az OpenSSL által használt algoritmusokat. Ha részletes hibajelentéseket szeretnénk, az `SSL_load_error_strings()` függvénnyel a hibaüzeneteket is be kell töltenünk, máskülönben az OpenSSL hibakódjait nem tudnánk a hibaüzenetekre leképezni.

A hibakiíráshoz is létrehozunk egy objektumot. Az OpenSSL a bemenet és kimenet kezeléséhez egy elvonatkoztatott BIO nevű objektumot használ, amivel a programozó számára lehetővé válik, hogy különféle I/O-csatornákat (foglalatok, terminál, memória stb.) ugyanazokkal a függvényekkel kezeljen, mindössze a megfelelő BIO objektumot kell választania. Ebben az esetben egy `stderr`-hez csatolt BIO objektumot hozunk létre, amelyet a hibák kiírására használunk.

Ha olyan ügyfelet vagy kiszolgálót írsz, amely képes az ügyfél hitelesítésére, be kell töltened a saját nyilvános, illetve titkos kulcspárodat és a hozzá kapcsolódó tanúsítványt. A tanúsítvány titkosítatlanul tárolódik és az `SSL_CTX_use_certificate_chain_file()` függvény tölti be a CA tanúsítványokkal együtt – létrehozva a tanúsítványláncot. A titkos kulcs az `SSL_CTX_use_PrivateKey_file()` függvénnyel tölthető be. Biztonsági okokból a titkos kulcsot sokszor jelszóval védik. Ha ez a helyzet, a vezérlés a jelszóbekérő visszahívó függvényre adódik (az `SSL_CTX_set_default_passwd_cb()`-t kell beállítani).

Amennyiben a gépet, amelyhez kapcsolódsz, hitelesíteni fogod, az OpenSSL-nek tudnia kell, mely tanúsítványhitelesítőkben bízol meg. A CA-k az `SSL_CTX_load_verify_locations()` hívással tölthetők be.

Az erős biztonság érdekében az SSL-nek jó minőségű véletlen számokra van szüksége. Általában az alkalmazás feladata a véletlenszám-előállító kezdőértékét megadni. Az OpenSSL azonban – amennyiben lehetséges – a ***/dev/urandom*** eszközt használja erre a célra. A /dev/urandom a szabványos Linux-rendszer része, ezért ez ügyben semmit nem kell tennünk, ami nagyon kényelmes, mert a véletlen számok gyűjtése bonyolult és könnyen elrontható feladat. Ne feledd, ha nem Linuxot használsz, egy bizonyos ponton hibaüzenetet kaphatsz, mert a véletlenszám-előállító nem kapott kezdőértéket. Az OpenSSL rand(3) súgóoldala többet mond erről.

1. lista initialize_ctx()

```
SSL_CTX *initialize_ctx(keyfile,password)
  char *keyfile;
  char *password;
  {
    SSL_METHOD *meth;
    SSL_CTX *ctx;

    if(!bio_err){
      /* A rendszer elıkØsz tØse */
      SSL_library_init();
      SSL_load_error_strings();

      /* A hibaki r  k rnyezet */

    bio_err=BIO_new_fp(stderr,BIO_NOCLOSE);
    }

    /* A SIGPIPE kezelı beÆll tÆsa */
    signal(SIGPIPE,sigpipe_handle);

    /* A k rnyezet nk lØtrehozÆsa*/
    meth=SSLv23_method();
    ctx=SSL_CTX_new(meth);

    /* A kulcsaink Øs tanœs tvÆnyaink
    bet ltØse */

if(!(SSL_CTX_use_certificate_chain_file(ctx,
      ➥keyfile)))
      berr_exit ("A tanœs tvÆnyfÆjl nem
                  ➥olvashat ");

    pass=password;
    SSL_CTX_set_default_passwd_cb(ctx,
      ➥password_cb);
    if(!(SSL_CTX_use_PrivateKey_file(ctx,
      keyfile,SSL_FILETYPE_PEM)))
      berr_exit("A kulcsfÆjl nem
                ➥olvashat ");

    /* A megb zhat  CA-k bet ltØse*/
    if(!(SSL_CTX_load_verify_locations(ctx,
       ➥CA_LIST,0)))
       berr_exit("A CA-k listÆja nem
                  ➥olvashat ");
#if (OPENSSL_VERSION_NUMBER < 0x0090600fL)
    ➥SSL_CTX_set_verify_depth(ctx,1);
#endif

    return ctx;
  }
```

Az ügyfél

Miután az ügyfél előkészítette az SSL-környezetet, készen áll a kiszolgálóhoz történő kapcsolódáshoz. Az OpenSSL megköveteli tőlünk, hogy a TCP-kapcsolatot magunk hozzuk létre az ügyfél és a kiszolgáló között, majd a TCP-foglalatot használjuk SSL-foglalat létrehozására. A kényelem kedvéért a TCP-kapcsolat létrehozását az elkülönített `tcp_connect()` függvény végzi (amely itt nem látható, de a letölthető forrásban benne van).

Miután a TCP-kapcsolat létrejött, létrehozzuk a kapcsolatot kezelő SSL-objektumot, amelyet a foglalathoz kell csatolni. Nem közvetlenül csatoljuk hozzá, hanem létrehozunk egy BIO objektumot, amely a foglalatot használja, tehát az SSL-objektumot a BIO-hoz csatoljuk.

Ez az elvonatkoztatási réteg lehetővé teszi, hogy a foglalatokon kívül más csatornákon keresztül is használhasd az OpenSSL-t, feltéve, hogy akad megfelelő BIO-d. Például az OpenSSL próbaprogramjainak egyike az SSL-ügyfelet és -kiszolgálót a memórián keresztül kapcsolja össze. Sokkal gyakorlatiasabb alkalmazás lehet olyan protokollok támogatása, amelyek nem érhetők el foglalatokon keresztül, így például SSL-t a soros vonalon keresztül futtathatsz.

Az SSL-kapcsolat első lépése az SSL-kézfogás végrehajtása. A kézfogás hitelesíti a kiszolgálót (esetleg az ügyfelet is), és meghatározza a későbbi adatforgalom védelmét biztosító kulcsokat. Az `SSL_connect()` hívás hajtja végre az SSL-kézfogást. Mivel tömbös foglalatokat használunk, az `SSL_connect()` nem tér vissza, amíg a kézfogás nem fejeződik be vagy hiba nem történik. Az `SSL_connect()` 1-et ad vissza siker, és 0-t vagy negatív számot hiba esetén. A hívás így néz ki:

```
/* Kapcsol dÆs a TCP- foglalathoz */
sock=tcp_connect(host,port);

/* Kapcsol dÆs az SSL- foglalathoz */
ssl=SSL_new(ctx);
sbio=BIO_new_socket(sock,BIO_NOCLOSE);
SSL_set_bio(ssl,sbio,sbio);
if(SSL_connect(ssl)<=0)
  berr_exit("SSL kapcsol dÆsi hiba");
if(require_server_auth)
  ➥check_cert(ssl,host);
```

Amikor a kiszolgálóhoz SSL-kapcsolatot kezdeményezünk, ellenőriznünk kell a kiszolgáló tanúsítványláncát. Az OpenSSL elvégzi az ellenőrzések egy részét, de a másik rész sajnos alkalmazásfüggő, ezért azokat nekünk magunknak kell elvégeznünk. Példaprogramunk fő tevékenysége a kiszolgáló „személyazonosságának” ellenőrzése – ezt a *2. listán* olvasható `check_cert` függvény végzi.

Miután meggyőződtél róla, hogy a kiszolgáló tanúsítványlánca érvényes, ellenőrizned kell, hogy az éppen vizsgált tanúsítvány arra a kiszolgálóra vonatkozik-e, amelyik felmutatta. A legtöbb esetben ez azt jelenti, hogy a kiszolgáló DNS neve megjelenik a tanúsítványban, vagy a ***Jogosult neve*** (Subject Name) mező ***Általános név*** (Common Name) részében, vagy a tanúsítvány kiterjesztésében. Bár minden egyes protokoll a kiszolgáló azonosságának ellenőrzését kissé eltérő módon oldja meg, az RFC 2818 tartalmazza az SSL/TLS-en keresztül megvalósított HTTP szabályait. Ha nincs okod más megoldás alkalmazására, kövesd az RFC 2818-at.

Mivel a legtöbb tanúsítvány a tartománynevet még mindig a ***Közönséges*** név mezőben tartja, és nem a kiterjesztésben, csak a Közönséges név ellenőrzését mutatjuk be. Az `SSL_get_peer_certificate()` segítségével egyszerűen megszerezzük a kiszolgáló tanúsítványát és összehasonlítjuk a közönséges nevet azzal a gépnévvel, amelyhez csatlakozunk. Ha a két név nem egyezik, akkor valami nincs rendben, és kilépünk.

A 0.9.5-változat előtt az OpenSSL ki volt téve a tanúsítvány-kiterjesztéses támadásnak. Ennek megértéséhez vegyük az

2. lista check_cert() függvény

```
void check_cert(ssl,host)
  SSL *ssl;
  char *host;
  {
    X509 *peer;
    char peer_CN[256];


if(SSL_get_verify_result(ssl)!=X509_V_OK)
      ➥berr_exit("A tanœs tvÆny nem
                  ➥ellenırizhetı");

    /* Ellenırizz k a tanœs tvÆnylÆncot!
       A lÆnc
       hosszœsÆgÆt  nmßk dıen ellenırzi az
       OpenSSL, ha beÆll tjuk az ellenırzØsi
       mØlysØget a k rnyezetben. */

    /*Az Æltalænos nØv ellenırzØse */
    peer=SSL_get_peer_certificate(ssl);
    X509_NAME_get_text_by_NID
      ➥(X509_get_subject_name(peer),
      ➥NID_commonName, peer_CN, 256);
    if(strcasecmp(peer_CN,host))
    err_exit
      ("Az Æltalænos nØv nem egyezik a
        ➥gØpnØvvel. ");
  }
```

ábrán látható esetet, ahol a kiszolgáló hitelesítését Béla írta alá. Béla nincs a CA-id között, de az ő tanúsítványát aláírta egy olyan CA, amiben megbízol.

Ha elfogadod ezt a tanúsítványt, nagy bajba kerülhetsz. Az a tény, hogy a CA aláírta Béla tanúsítványát, azt jelenti, hogy a CA elhiszi Béláról, hogy ő az, akinek állítja magát, de nem jelenti azt, hogy Béla megbízható. Ha Bélával akarsz üzletelni, ez rendben van, viszont ez nem sokat ér, ha Aladárral szeretnéd ugyanezt megtenni, és Béla (akiről sosem hallottál) áll jót érte. Eredetileg ez ellen a támadás ellen az egyetlen védekezési mód az volt, hogy megkötötték a tanúsítványláncok hosszát, így tudható volt, hogy a vizsgált tanúsítványt a CA írta alá. Az X.509 3. változata lehetővé teszi, hogy a CA bizonyos tanúsítványokat úgy címkézzen meg, mint más CA-k. Így egy CA-nak egyetlen gyökere lehet, amely azután egy csomó al-CA-nak nyújthat tanúsítványt. Az OpenSSL újabb változatai (a 0.9.5-nél frissebbek) figyelik ezeket a kiterjesztéseket, ezért akár ellenőrzöd a lánc hosszát, akár nem, védve vagy e támadás ellen. A 0.9.5 előtti változatok egyáltalán nem ellenőrzik a kiterjesztéseket, ezért – amennyiben ilyen régi változatot használsz – neked kell a lánc hosszának az ellenőrzését elvégezned. A 0.9.5-nek gondjai akadhatnak az ellenőrzéssel, ha tehát ezt a változatot használod, fontold meg a frissítést. Az `#if` utáni sor régebbi változat használata esetén az `initialize_ctx()` kódjában ellenőrzi a lánc hosszát. Az `SSL_CTX_set_verify_depth()` függvényt használjuk a lánchossz ellenőrzésének kikényszerítésére. Mindent összevetve nagyon ajánlatos a 0.9.6 változatra frissíteni, különösen azért, mert a hosszú (de szabályosan felépített) láncok egyre népszerűbbek. A legújabb és legjobb OpenSSL-változat jelenleg a 0.9.66.

A 3. listán (23. CD Magazin/OpenSSL könyvtár) látható kód segítségével írunk HTTP-kérést. A bemutató kedvéért egy többé-kevésbé bedrótozott HTTP-kérést használunk, amely a `REQUEST_TEMPLATE` változóban van. Mivel a gép, amelyhez csatlakozunk, változhat, ki kell töltenünk a ***Gép*** (Host) fejlécet, amit az `snprintf()` végez el. Ezután az `SSL_write()` elküldi az adatokat a kiszolgálónak. Az `SSL_write` API-ja nagyjából ugyanaz, mint a `write()`, kivéve hogy SSL-objektumot adunk át és nem fájlleírót.

A tapasztalt TCP-programozók észrevehették, hogy ha a visszatérési érték nem egyenlő a kiírni próbált értékkel, a kiírás körbejárása helyett hibát jelzünk. Tömbös módban az `SSL_write()` minden vagy semmi elven működik, a hívás nem tér vissza, amíg az adatot ki nem írta vagy hiba nem történt, míg a `write()` esetleg csak az adatok egy részét írja ki. Az `SSL_MODE_ENABLE_PARTIAL_WRITE` kapcsoló (itt nem használjuk) engedélyezi a részleges írást, ebben az esetben szükség van a ciklusra.

A régi stílusú HTTP/1.0 használatánál a kiszolgáló átküldi a választ és lezárja a kapcsolatot. A későbbi változatoknál bevezették az állandó kapcsolatot, amelynél több, sorban egymást követő tranzakció egy kapcsolatot használ. Az egyszerűség és a kényelem kedvéért mi nem fogunk ilyet használni. Elhagyjuk az ezeket engedélyező fejlécet, ennek hatására a kiszolgáló a válasz végén lezárja a kapcsolatot, ami gyakorlatilag azt jelenti, hogy a fájl végéig folyamatosan kell olvasni, és ez nagyban leegyszerűsíti a dolgokat.

Az `OpenSSL` az adatok beolvasására az `SSL_read()` API-hívást használja, ahogyan a 4. lista (23. CD Magazin/OpenSSL könyvtár) mutatja. Akárcsak a `read()`-nél, egyszerűen választunk egy megfelelő méretű tárolót és átadjuk az `SSL_read()`-nek. A tároló mérete nem túl fontos. Az `SSL_read()` a `read()`-hez hasonlóan az összes elérhető adatot visszaadja, akkor is, ha az kevesebb, mint a kért mennyiség. Egyébként, ha nincs elérhető adat, az olvasóhívás útja elzáródik.

A BUFSIZZ választása befolyásolja a hatékonyságot, de nem olyan módon, mint amikor egyszerűen egy foglalatból olvasunk. Abban az esetben minden `read()` híváskor át kell váltani a rendszermagba. Az átváltás drága művelet, ezért a programozók nagy tárolókat szoktak választani, hogy csökkentsék a szükséges átváltások számát. Amikor azonban az SSL-t használjuk, a `read()`-hívások száma és így az átváltásoké is inkább a kiírt adatrekordok számától függ, semmint az `SSL_read()`-hívások számától.

Ha például az ügyfél kiír egy 1000 bájtos rekordot, és mi 1 bájtonként hívjuk meg az `SSL_read()` függvényt, akkor az első `SSL_read()`-hívás alkalmával az egész rekord beolvasódik, és a többi hívás csak kiolvassa az értékeket az SSL-tárolóból. Ezért a tároló mérete, ellentétben a szokásos foglalatokkal, SSL használata esetén kevésbé lényeges. Ha az adatok sok kis rekord sorozataként íródnak ki, egyszerre egyetlen `read()`-hívással beolvashatod őket. Ilyenkor az OpenSSL `SSL_CTRL_SET_READ_AHEAD` kapcsolója használható.

Ne feledd el kiértékelni az `SSL_get_error()` visszatérési értékét! A szokásos foglalatoknál minden negatív szám (általában -1) hibát jelez, és ekkor az `errno` értékét megvizsgálva megtudhatjuk, mi történt. Természetesen az `errno` itt nem használható, mert csak a rendszerhibákat mutatja, és minket az SSL-hibák érdekelnek. Az `errno` alkalmazása nagy körültekintést is igényel ahhoz, hogy többszálú környezetben is biztonságos legyen.

Az `errno` helyett az OpenSSL az `SSL_get_error()` hívást biztosítja. Ez a hívás lehetővé teszi a visszatérési érték vizsgá-

latát és az esetleges hiba mibenlétének megismerését. Ha a visszatérési érték pozitív, akkor adatokat olvastunk be, amelyeket egyszerűen kiírunk a képernyőre. Egy valódi ügyfél természetesen értelmezné a HTTP-választ, és vagy megjelenítené az adatokat (például egy weboldalt), vagy lemezre mentené. Az OpenSSL szempontjából azonban teljesen mindegy, hogy mi lesz az adatok sorsa, ezért nem foglalkozunk vele. Ha a visszatérési érték nulla, nem jelenti azt, hogy nincs elérhető adat – ez esetben az utunk el lett volna zárva, ahogyan fent említettük. Ez inkább azt tükrözi, hogy a foglalat zárva van és olvasásra soha semmilyen adat nem lesz elérhető. Így kilépünk a ciklusból.

Ha a visszatérési érték negatív, valamilyen hiba történt. Kétféle hibatípussal kell számolnunk: közönséges hibákkal és idő előtti lezárásokkal. A hiba típusát az `SSL_get_error()` hívással állapítjuk meg. A hibakezelés ügyfelünkben nagyon kezdetleges, a legtöbb hibát a `berr_exit()` meghívásával csak kiírjuk, amely ezután kilép a programból. Az idő előtti lezárásokat külön kell kezelnünk.

A TCP a FIN-csomagrészt használja annak jelzésére, hogy a küldő minden adatot elküldött. Az SSL 2. változata az SSL-kapcsolat lezárására mindkét félnek megengedte a TCP FIN küldését, ami úgynevezett csonkolásos támadásra adott lehetőséget. A támadó elhitethette, hogy az üzenet rövidebb a valóságosnál, egyszerűen a TCP FIN-t meghamisítva. Ha az áldozat más módon nem tudhatta meg az üzenet várható hosszát, könnyen azt hihette, hogy a hossz rendben volt.

Ennek a biztonsági kockázatnak a kivédésére vezették be az SSLv3-ban a ***close_notify*** figyelmeztetést, ami egy SSL-üzenet (emiatt biztonságos), de nem része magának az adatfolyamnak, ezért az alkalmazás nem látja. A ***close_notify*** elküldése után semmilyen adatot nem szabad átvinni. Ezért amikor az `SSL_read()` nullával tér vissza, amely azt jelzi, hogy a foglalatot lezárták, ez valójában azt jelenti, hogy a ***close_notify*** figyelmeztetés megérkezett. Ha az ügyfél FIN-t kap a ***close_notify*** előtt, az `SSL_read()` hibával tér vissza. Ezt a jelenséget nevezik idő előtti lezárásnak. Az ügyfél minden idő előtti lezárásnál dönthet úgy, hogy kiírja a hibát és kilép. Ezt a viselkedést vonja maga után az SSLv3-szabvány. Az idő előtti lezárás sajnos elég gyakori hiba, különösen az ügyfeleknél. Ezért hacsak nem akarsz állandóan hibákat jelenteni, gyakran figyelmen kívül kell hagynod őket – kódunk külön figyel erre a hibára: jelenti az idő előtti lezárást az `stderr`-en, de nem lép ki.

Ha hiba nélkül olvastuk el a választ, a kiszolgálónak egy ***close_notify*** figyelmeztetést kell küldenünk. Ezt végzi el az `SSL_shutdown()` API hívás. A kiszolgálónál részletesebben fogjuk tárgyalni az `SSL_shutdown()`-t, de az alapötlet egyszerű: a visszatérési érték 1, ha a leállítás teljes, 0, ha a leállítás nem teljes, és -1, ha hiba történt. Mivel már megkaptuk a kiszolgáló ***close_notify*** üzenetét, az egyetlen dolog, amivel gond lehet, ha nem tudjuk elküldeni a saját ***close_notify*** üzenetünket. Egyébként az `SSL_shutdown()` sikeres lesz (1-et ad vissza).

Végül meg kell semmisítenünk a különféle objektumokat, amelyeknek helyet foglaltunk. Mivel a program kilépésre készül, az objektumok felszabadítása nem lenne feltétlenül szükséges, de egy általánosabb programban szükséges volna.

A kiszolgáló

Webkiszolgálónk néhány különbségtől eltekintve az ügyfél tükörképe. Először a `fork()` használatával elágaztatjuk a programot, hogy a kiszolgáló több ügyfelet is kiszolgálhasson. Másodszor: az OpenSSL BIO API-ját használjuk az ügyfél kérésének sorról sorra történő beolvasásához és a válasz kiírásához. Végül a kiszolgáló lezárási folyamata bonyolultabb.

A több ügyfelet kezelni képes kiszolgáló írásának Linux alatt az a legegyszerűbb módja, hogy minden kapcsolódó ügyfélhez egy új kiszolgáló folyamatot indítunk. Ezt a `fork()` hívás segítségével tesszük meg, miután az `accept()` visszatért. Minden egyes új folyamat függetlenül fut, és egyszerűen kilép, ha végzett az ügyfél kiszolgálásával. Bár nagy forgalmú webkiszolgálóknál a megközelítés nagyon lassú lehet, esetünkben teljesen elfogadható.

A kiszolgáló fő fogadóciklusát az 5. listán (23. CD Magazin/OpenSSL könyvtár) láthatjuk.

Az elágazás és az SSL-objektum létrehozása után a kiszolgáló meghívja az `SSL_accept()` függvényt, amely a kézfogás kiszolgálóoldali részéért felelős. Akárcsak az `SSL_connect()`, a tömbös foglalatok használata miatt az `SSL_accept()` is lezárja a csatornát, amíg a kézfogás be nem fejeződik. Ezért az `SSL_accept()` csak akkor tér vissza, ha a kézfogás befejeződött vagy hiba lépett fel. Az `SSL_accept()` 1-et ad vissza siker esetén, és 0-t vagy negatív számot, ha hiba történt.

Az OpenSSL BIO objektumai bizonyos mértékig egymásra halmozhatók. Ezért az SSL-objektumot becsomagolhatjuk egy BIO-ba (az `ssl_bio` objektumba), és ezt a BIO-t csomagoljuk tovább egy tárazott BIO objektumba a következő módon:

```
io=BIO_new(BIO_f_buffer());
ssl_bio=BIO_new(BIO_f_ssl());
BIO_set_ssl(ssl_bio,ssl,BIO_CLOSE);
BIO_push(io,ssl_bio);
```

Ez lehetővé teszi a számunkra, hogy tárazott olvasási és írási műveleteket hajtsunk végre az SSL-kapcsolaton keresztül a BIO_* függvények és az új `io` objektum segítségével. E ponton felvetődhet a kérdés, hogy mire jó mindez.

Elsősorban a programozás lesz kényelmesebb: a programozó számára lehetővé válik, hogy az SSL-rekordok helyett természetes egységekkel (sorokkal és karakterekkel) dolgozzon.

A kérés

A HTTP-kérés egy kéréssorból, az ezt követő fejlécsorokból és az esetleges törzsből áll. A fejlécsorok végét üres sor jelzi (azaz egy CRLF pár, bár a hibás ügyfelek ehelyett néha egy LF karaktert küldenek). A kéréssort és a fejlécet a legkényelmesebb soronként beolvasni, amíg üres sorral nem találkozunk. Az `OpenSSL_BIO_gets()` hívás segítségével ez megtehető, lásd a 6. listát (23. CD Magazin/OpenSSL könyvtár). A `BIO_gets()` hívás hasonlóan működik, mint az `sdtio fgets()` hívása. Fog egy tetszőleges nagyságú tárolót és egy hosszúságot, és az SSL-ről beolvas egy sort a tárolóba. Az eredmény mindig `NULL` karakterre végződik (de benne van a végső `LF`). Ezért az adatokat egyszerűen addig olvassuk be soronként, amíg olyan sorral nem találkozunk, amely csak `LF`-et vagy `CRLF`-et tartalmaz.

Mivel állandó méretű tárolót használunk, előfordulhat, bár nem valószínű, hogy túl hosszú sort kapunk. Ebben az esetben a hosszú sor kettébomlik. Abban a nagyon valószínűtlen esetben, ha a szétbontás közvetlenül a `CRLF` előtt történik, a következő beolvasott sor az előző sorból származó `CRLF`-et fogja tartalmazni. Ez esetben azt fogjuk hinni, hogy a fejléc idő előtt véget ért. Az igazi webkiszolgálók erre az esetre is figyelnek, de itt nem éri meg nekünk a fáradságot. Jegyezzük meg, hogy bármilyen nagy a bejövő sor hossza, nem léphet fel tártúlcsordulás. A legrosszabb esetben félreértelmezzük a fejlécet. Igazából semmit nem kezdünk a HTTP-kéréssel, egyszerűen

beolvassuk, utána elfelejtjük. Az igazi alkalmazások elolvasnák a kéréssort és a fejlécet, megnéznék, hogy van-e törzs, és azt is elolvasnák.
A következő lépés a HTTP-válasz kiírása és a kapcsolat lezárása:

```
if((r=BIO_puts
  ➥(io,"HTTP/1.0 200 OK\\r\\n"))<0)
  err_exit("´rÆshiba");
if((r=BIO_puts
  ➥(io,"Server: EKRServer\\r\\n\\r\\n"))<0)
  err_exit("´rÆshiba");
if((r=BIO_puts
  ➥(io,"KiszolgÆl  tesztoldala\\r\\n"))<0)
  err_exit("´rÆshiba");

if((r=BIO_flush(io))<0)
  err_exit("Hiba a BIO ki r tØse k zben");
```

Vegyük észre, hogy `BIO_puts()` hívást használtunk az `SSL_write()` helyett, ami lehetővé teszi, hogy soronként írjuk ki a választ, de az egész válasz egyetlen SSL-rekordba kerüljön. Ez az SSL-rekord összeállításának költségei miatt fontos. A sértetlenség ellenőrzése és a titkosítás jelentős erőforrásokat köt le, emiatt jó ötletnek látszik olyan nagy rekorddal dolgozni, amilyennel csak lehet.
Érdemes megemlíteni a fent említett tárazott kiírás néhány finomságát. Először is lezárás előtt ki kell ürítened a tárat. Az SSL-objektumnak fogalma sincs arról, hogy egy BIO-t rétegeztél fölé, így ha az SSL-kapcsolatot megsemmisíted, az adatfolyam utolsó darabja ott marad a tárban. A `BIO_flush()` megoldja a gondot. Alapértelmezés szerint az OpenSSL a BIO-khoz 1024 bájtos átmeneti tárakat használ. Mivel az SSL-rekordok akár 16 KB hosszúak is lehetnek, az 1024 bájtos átmeneti tárak használata nagymértékű töredezettséghez (és emiatt teljesítménycsökkenéshez) vezethet. A `BIO_ctrl()` API-hívást használhatod a tár méretének növelésére.
A válasz átvitele után el kell küldenünk a ***close_notify*** üzenetet. Akárcsak az előbb, most is az `SSL_shutdown()` segítségével végezzük el a feladatot. A helyzet sajnos egy kissé bonyolultabb, amikor a kiszolgáló zárja le előbb a kapcsolatot. Az első `SSL_shutdown()` hívás elküldi a ***close_notify***-t, de azt nem keresi a másik oldalon, ezért azonnal visszatér, de 0 értékkel, ami azt jelenti, hogy a lezárási folyamat nem ért még véget. Az alkalmazásnak kell ismét az `SSL-shutdown()`-t meghívnia. Kétféleképpen járhatunk el: mondhatjuk azt, hogy megkaptuk a teljes HTTP-kérést, amely bennünket érdekel. Semmi mással nem törődünk. Emiatt az sem érdekel minket, hogy az ügyfél küld-e ***close_notify***-t vagy sem. A másik lehetőség, hogy a protokoll előírásait szigorúan vesszük, és mástól is ezt várjuk el, ezért megköveteljük a ***close_notify***-t.
Az első hozzáállás könnyű életet biztosít. Meghívjuk az `SSL_shutdown()`-t, elküldjük a ***close_notify***-t és azonnal kilépünk, függetlenül az ügyfél viselkedésétől. Ha a második utat követjük (ahogy példánkban a kiszolgáló), az élet sokkal bonyolultabb, mert az ügyfelek gyakran helytelenül viselkednek. Az első gond, amellyel szembe kell néznünk, hogy az ügyfelek sokszor egyáltalán nem küldenek ***close_notify***-t; egyes ügyfelek (például az IE bizonyos változatai) a HTTP-válasz kézhezvétele után azonnal lezárják a kapcsolatot. Amikor elküldjük a ***close_notify***-t, a túloldal egy TCP RST-csomagrészt küldhet. Ebben az esetben a program egy SIGPIPE jelzést kap. Védekezésképpen az `initialize_ctx()` függvényben beállítunk egy egyszerű `SIGPIPE`-ot kezelő programrészt.

A második gond, hogy az ügyfél a ***close_notify***-t válaszul a mi ***close_notify***-unkra esetleg nem azonnal küldi. A Netscape bizonyos változatai azt várják, hogy először te küldj egy TCP FIN-t. Ezért hívjuk meg a `shutdown(s,1)`-et a második `SSL_shutdown()`-hívás előtt. Ha a `shutdown()`-t az 1 módértékkel hívjuk meg, elküldi a FIN-t, de a foglalatot nyitva hagyja olvasásra. A kiszolgáló leállításának kódja a 7. listán (23. CD Magazin/OpenSSL könyvtár) olvasható.

Ami kimaradt

Ebben a cikkben csak az OpenSSL felszínét karcolhattuk. Következzék a további lehetséges témák (nem teljes) felsorolása. A kiszolgáló tanúsítványainak és gépnevének összevetésére sokkal kifinomultabb módszer az X.509 subjectAltName kiterjesztés használata. Az összehasonlításhoz ezt a kiterjesztést ki kell szedni a tanúsítványból, és össze kell vetni a gépnévvel. Továbbá jó lenne a gépneveket a tanúsítványokban szereplő helyettesítő karakterekkel megadott nevekhez hasonlítani. Figyeljük meg, hogy ezek az alkalmazások hiba esetén egyszerűen kilépnek. Az igazi alkalmazások természetesen képesek felismerni a hibát és kilépés helyett jelezni a felhasználónak, vagy a naplóba írni.
A következő cikkben számos fejlett OpenSSL-sajátosságot tárgyalunk, többek között a kapcsolat folytatását, a többutas és nem tömbös I/O- és ügyfélhitelesítést.

Köszönetnyilvánítás

Köszönettel tartozom a következő személyeknek, amiért segítettek az OpenSSL-ben és lektorálták a cikkemet: ***Lisa Dusseault***, ***Steve Henson***, ***Lutz Jaenicke*** és ***Ben Laurie***.

Eric Rescorla
1993 óta foglalkozik az Internet biztonságával. Ő a szerzője az SSL and TLS: Designing and Building Secure Systems (Addison-Wesley, 2001) című könyvnek.

Kapcsolódó címek

A cikk egyes részeit a könyvemből vettem át (SSL and TLS: Designing and Building Secure Systems, copyright Addison-Wesley, 2001, ISBN 0-201-615980-3). A könyv részletesen tárgyalja az SSL protokollt és alkalmazásait. További adatokért lásd a ➲ http://www.rtfm.com/sslbook weblapot.

Az itt bemutatott programok forráskódja a szerző weblapjáról letölthető: ➲ http://www.rtfm.com/openssl-examples. A forráskódra BSD stílusú felhasználói szerződés vonatkozik.

Az OpenSSL a ➲ http://www.openssl.org-ról tölthető, ahol a leírás is elérhető.

Az SSLv2 és az SSLv3 leírása megtekinthető a ➲ http://www.netscape.com/eng/security/SSL_2.html és a home.netscape.com/eng/ssl3/index.html weblapokon. A TLS (RFC 2246) és a HTTPS (RFC 2818) leírása a ➲ http://www.ietf.org/rfc/rfc2246.txt és a ➲ http://www.ietf.org/rfc/rfc2818.txt címeken lelhető fel.