# VIA PadLock – Boszorkányosan gyors titkosítás

A VIA olcsón beszerezhető processzora támogatja az Advanced Encryption Standardet, így segítségével kiemelkedő szintű titkosítást végezhetünk, méghozzá hardveres szintű sebességgel.

Aki használt már titkosítást, az valószínűleg hamar észrevette, hogy a művelethez nem csekély processzorteljesítményre van szükség. A régebbi rendszereken például a titkosított fájlrendszerek használata lassabb fájlműveleteket eredményez, az újabb rendszereken pedig legalábbis magasabb processzorterheléssel kell számolnunk. A hálózati forgalom ***IPsec*** alapú titkosítása szintén lelassítja a dolgokat, és sokszor még egy hétköznapi, 100 Mbps sebességű hálózaton is észlelhető a teljesítmény visszaesése.

- A titkosítás/teljesítmény kompromisszum kezelésére többféle út kínálkozik:
- Nem használunk titkosítást: látszólag a legolcsóbb megoldás, ám hosszú távon rendkívül drágává is válhat.
- A lassúbbodás elfogadása: a legjellemzőbb megoldás.
- Különálló titkosításgyorsító eszköz használata: például ***PCI*** buszra illeszkedő kártya, amely sajnos nem javít annyit a helyzeten, mint várnánk, ugyanis az adatoknak a feltétlenül szükségesnél többször kell megjárniuk a ***PCI*** buszt.
- ***VIA PadLock*** technológiát használó processzor használata. De vajon mi az a ***VIA PadLock***? Rögtön kiderül.

## VIA PadLock

Nem is oly rég a ***VIA*** egyszerű és egyben vitatható ötlettel állt elő: válasszunk ki néhány kriptográfiai algoritmust, majd drótozzuk be őket a processzorba. Létrejött tehát egy ***i686*** osztályú processzor, amely néhány új, kifejezetten titkosítási célt szolgáló utasítást is képes megérteni. A technológia a ***VIA PadLock*** nevet kapta, míg maga a processzor teljes mértékben kompatibilis az ***AMD Athlon*** és az ***Intel Pentium*** lapkákkal.
A rendelkezésre álló ***PadLock*** szolgáltatások körét a gép processzorának változata határozza meg. A processzorváltozatokat általában család-modell-fokozat hármassal jelölik. Az ***i686*** osztályú processzorok mindegyike a 6-os családba tartozik. Ha a modellszám 9-es, akkor a processzor ***Nehemiah***, ha 10-es, akkor pedig ***Esther*** maggal rendelkezik. A fokozatok az egyes modellek megújított változatai. A processzor változatát a ***/proc/cpuinfo*** fájlban találjuk meg.
A ***Nehemiah 3***-as fokozata, illetve az újabb példányok ***elektromos zaj alapú véletlenszám-előállítóval (random number generator, RNG)*** rendelkeznek, ami meglehetősen jó véletlen értékeket képes előállítani különféle célokra. Az ***RNG*** elérésére az `xstore` parancs szolgál. Az ***Intel*** és az ***AMD*** processzorokhoz hasonlóan a ***VIA*** processzorok véletlenszám-előállítóját is a `hw_random` illesztőprogram támogatja.
A ***Nehemiah 8***-as és újabb fokozatai már két független ***RNG***-t tartalmaznak, valamint az ***Advanced Cryptography Engine***-nel ***(speciális kriptográfiai motor, ACE)*** bővültek. Az ***ACE*** az ***Advanced Encryption Standard (speciális titkosítási szabvány, AES)*** algoritmus alkalmazásával, háromféle szabványos kulcshosszal – 128, 192 és 256 bit – négyféle módban képes adatokat titkosítani és visszafejteni. Az üzemmódok a következők:

- Elektronikus kódkönyv (electronic codebook, ECB)
- Titkosítási blokkláncolás (cipher block chaining, CBC)
- Titkosítási visszacsatolás (cipher feedback, CFB)
- Kimeneti visszacsatolás (output feedback, OFB)

Az ezeknek megfelelő utasítások az `xcryptecb`, az `xcryptcbc` stb. A továbbiakban ezeket közös névvel `xcrypt`-nek fogom nevezni, az egyes módokhoz tartozó parancsokat nem fogom használni.
Az ***Esther*** mag 0-s és újabb fokozatai örökölték a ***Nehemiah*** két ***RNG*** egységét. Az ***ACE*** kibővült a ***számláló (counter, CTR)*** mód támogatásával, valamint az ***üzenethitelesítő kódok (Message Authentication Code, MAC)*** számításának lehetőségével. Itt két újabb rövidítéssel kellett megismerkednünk, ezek a ***PHE*** és a ***PMM***. A ***PadLock Hash Engine (PadLock kivonatoló motor, PHE)*** adott bemeneti blokk kriptográfiai célú kivonatának elkészítésére alkalmas; az ***SHA1*** vagy az ***SHA256*** algoritmussal dolgozik. Az ide kötődő parancs az `xsha`.
A ***PadLock Montgomery Multiplier (PMM, PadLock Montgomery szorzó)*** az aszimmetrikus, más szóval nyilvános kulcsú titkosítási eljárások során a legszélesebb körben használt parancsok egyikének felgyorsítását szolgálja; ez az $A^B$ ***mod M***, ahol az ***A***, a ***B*** és az ***M*** hatalmas nagy, általában 1024 vagy 2048 biten ábrázolható szám. Szolgáltatásait a `montmul` utasítással vehetjük igénybe.
Mint már utaltam rá, a továbbiakban leginkább az xcrypt utasításokról lesz szó. Az ismertetésre kerülő alapelvek túlnyomó részt a többi egységre is érvényesek, és az xcrypt csupán példaként szolgál. Sőt, a titkosítás kapcsán említett kifejezések és fogalmak a visszafejtés területén is megőrzik érvényüket.

## A PadLock használata

A külső, általában a ***PCI*** buszra csatlakozó kriptográfiai gyorsítókkal ellentétben a ***PadLock*** motor a processzor beépített része. Ennek köszönhetően jóval egyszerűbb a használata,

1. táblázat *Az OpenSSL teljesítményteszt eredménye 1,2 GHz-es órajelű VIA Nehemiah processzoron [kBps]*

| Típus | 16bájt | 64 bájt | 256 bájt | 1024 bájt | 8192 bájt |
|---|---|---|---|---|---|
| aes-128-ecb (szoftveres) | 11274,53 | 14327,79 | 14608,64 | 14672,55 | 14693,72 |
| aes-128-ecb (PadLock) | 66892,82 | 346583,52 | 910704,21 | 1489932,59 | 1832151,72 |
| aes-128-cbc (szoftveres) | 8276,27 | 12915,75 | 13264,13 | 13313,02 | 13322,92 |
| aes-128-cbc (PadLock) | 48542,30 | 241898,79 | 523706,28 | 745157,61 | 846402,90 |

hiszen nem kell a busz elérésével vagy éppen a megszakítások és az aszinkron műveletek lekezelésével foglalkozni. Egy-egy memóriablokkot titkosítani egy `xcrypt` utasítással ugyanolyan egyszerű, mint átmásolni a `movs` utasítással. Ezen a szinten a titkosítás majdnem oszthatatlan művelet. Az utasítás végrehajtása előtt a puffer a nyílt adatokat tartalmazza, néhány órajellel később, az utasítás végrehajtásának befejezése után pedig már a titkosítottakat. Ha a kért művelet egyetlen blokk feldolgozása, ami az ***AES*** algoritmus esetében 16 bájtot jelent, akkor a művelet teljesen oszthatatlan. Vagyis, a processzor nem szakítja meg a műveletet, és semmi másba nem kezd bele, amíg a titkosítással nem végzett. De mi történik, ha a puffer több gigabájtnyi titkosítandó adatot tartalmaz? Nem volna túl jó ötlet minden más műveletet leállítani a hosszan tartó titkosítás befejezéséig. Ilyenkor a processzor minden egyes 16 bájtos blokk után jogosult megszakítani a titkosítást. Elmenti a pillanatnyi állapotot, majd elvégzi a rá várakozó feladatokat, mint a megszakítások kezelése vagy az egyéb folyamatok futtatása. A titkosítási folyamat újraindításakor az utasítás elvégzése attól a ponttól folytatódik, ahol felfüggesztésre került. Ez az, amiért csak majdnem oszthatatlan a művelet: a hívó folyamat számára annak látszik, ám nagyobb fontosságú esemény megszakíthatja.
A pillanatnyi feldolgozási állapot és a futó folyamat regiszterei elmentésre kerülnek a memóriába, vagyis egyszerre több folyamat is végezhet titkosítást, nem kell adataik összekeveredésétől tartani. Ismétlem, mindez sokban hasonlít a memóriablokkoknak a `movs` utasítással végzett másolására.

## Mennyire gyors?

A ***VIA*** adatai szerint egy 1,2 GHz órajelű processzor maximális átviteli sebessége több mint 15 Gbps, vagyis majdnem 1,9 GBps. Az általam végzett teljesítménymérések igazolták, hogy valós alkalmazásoknál is láthatunk hasonló sebességeket, nemcsak a ***VIA*** marketinganyagaiban, ami rendkívül kellemes meglepetés volt.
A tényleges titkosítási sebesség két tényezőtől függ, a titkosítási módtól és az adatok igazításától. Az ***ECB*** a leggyorsabb, a legszélesebb körben használt ***CBC*** pedig nagyjából az ***ECB*** sebességének felét képes hozni. A ***PadLock*** megköveteli az adatok 16 bájtos határokra igazítását, ezért az igazítás nélküli adatokat először a megfelelő címekre kell mozgatni, ami némi időveszteséget okoz. Egyes esetekben az ***Esther*** processzorok képesek az adatok önműködő igazítására is, ám a sebességcsökkenés ekkor is fellép.
Az *1. táblázat* méréseim eredményeinek egy részét tartalmazza. Az eredményeket az ***OpenSSL*** teljesítményteszttel, 1,2 GHz-es ***VIA Nehemiah*** processzoron kaptam, a mennyiségek mértékegysége kBps.

Minél nagyobbak a blokkok, annál jobbak az eredmények, ugyanis eltűnnek az ***OpenSSL*** könyvtár által okozott többletterhelések. A 8 kB-os blokkok ***ECB*** módban végzett titkosítása körülbelül 1,7 GBps sebességgel lehetséges, míg ***CBC*** módban 800 MBps-ot kapunk.
A szoftveres titkosítással összehasonlítva a ***PadLock ECB*** módban ugyanazon a processzoron 120-szor gyorsabb, míg a ***CBC*** mód esetében 60-szoros az eltérés.
A gyorsulásnak köszönhetően az ***IPsec*** 100 Mbps sebességű hálózaton teljes értékűen, nagyjából 11 MBps sebességgel működik. A titkosított fájlrendszerek esetében hasonló sebességjavulásokat lehet tapasztalni. A ***Bonnie*** teljesítménytesztet egy ***UDMA100*** módban üzemelő ***Seagate Barracuda*** merevlemezzel futtatva, nyílt adatokkal 61543 kBps-os átviteli sebességet kapunk. ***PadLockot*** használva ez 49961 kBps-ra mérséklődik, míg tisztán szoftveres titkosítás használatakor csupán 10005 kBps-ot lehet elérni. A ***PadLock*** alkalmazásakor tehát a titkosítatlan átvitelhez képest csupán 20 százalékos teljesítménycsökkenéssel kell számolni, míg a szoftveres megoldásnál ez közel 85 százalék. A források között szerepel egy a teljesítménymérések eredményeit tartalmazó oldalamra mutató hivatkozás; ott további részletek és számok is megtalálhatók.

## Linux alatti támogatás

A linuxos támogatást eddig – az alábbi csomagokhoz – csak az `xcrypt` utasítás révén használható ***AES*** algoritmushoz készítettem el, ***Esther*** magos processzorhoz ugyanis még nem jutottam hozzá. Amint megkapom az új processzor egy példányát, szükség szerint meg fogom oldani a többi algoritmus támogatását is.

## Rendszermag

Ha a rendszermagnak szüksége van az ***AES*** algoritmusra, akkor alapesetben az `aes.ko` modult tölti be, amely az algoritmus szoftveres megvalósítása. Ha az ***AES***-t a ***PadLockkal*** akarjuk futtatni, akkor az `aes.ko` helyett a `padlock.ko` modult kell betöltenünk. Ezt kézzel, a `modprobe` paranccsal tehetjük meg, illetve az alábbi sort hozzáadva a */etc/modprobe.conf* fájlhoz:

```
alias aes padlock
```

Ezt követően minden alkalommal, amikor ***AES***-re lesz szüksége, a rendszermag a `padlock.ko` modult is betölti.
Foltok a rendszermag 2.6.5-ös és újabb változataihoz léteznek; lásd a források közötti linuxos ***PadLock*** oldalt. Az alapszintű illesztőprogram a 2.6.11-es rendszermag alapváltozatában, mindenféle foltozás nélkül is elérhető lesz.

## OpenSSL

Azok, akik elég merészek ahhoz, hogy az ***OpenSSL*** újabb, ***CVS*** alatti változatait használják, már rendelkeznek a ***PadLock*** támogatásával. Az ***OpenSSL 0.9.7***-es változatát futtatóknak meg kell foltozniuk és újra kell fordítaniuk a könyvtárat; illetve lehetséges, hogy terjesztésük csomagjai már eleve tartalmazzák a foltot, ilyen terjesztés például a ***SuSE Linux 9.2***. Azt, hogy saját ***OpenSSL*** példányunk rendelkezik-e ***PadLock*** támogatással, az alábbi egyszerű paranccsal vizsgálhatjuk meg:

```
$ openssl engine padlock
(padlock) VIA PadLock (RNG, ACE)
```

Az `(RNG, ACE)` helyett lehet, hogy `(no-RNG, no-ACE)` jelenik meg, ez azt jelenti, hogy bár ***OpenSSL***-ünk képes a ***PadLock*** támogatására, gépünk processzora nem nyújt ilyen szolgáltatást. Lehet, hogy csúf hibaüzenet jelzi, hogy nincs ilyen motor a gépünkön, ekkor el kell végeznünk az ***OpenSSL*** könyvtár frissítését vagy foltozását. A titkosítási műveleteik végrehajtására az ***OpenSSL***-t használó programoknak a ***PadLock*** támogatás használatához az úgynevezett ***EVP_interface***-t kell használniuk, valamint valamikor futásuk elején el kell végezniük a hardveres gyorsító kezdeti értékadását:

```
#include <openssl/engine.h>
int main ()
{
    [...]
    ENGINE_load_builtin_engines();
    ENGINE_register_all_complete();
    [...]
}
```

További részletek az ***OpenSSL*** leírásának ***evp(3) man*** oldalán találhatók. Például ***SuSE Linux 9.2*** alatt az ***OpenSSH***-nak van egy hasonló foltja, amivel gyorsabb hálózati ***scp***-átviteleket lehet végezni.

## Binutils

Ha a ***PadLockot*** saját programjainkban akarjuk használni, akkor a megfelelő utasítást név szerint is meghívhatjuk – mint például `xcryptcbc` –, de hexadecimális formában, közvetlenül is írhatjuk:

```
.byte 0xf3,0x0f,0xa7,0xd0
```

A régebbi fejlesztőeszközökkel való együttműködés biztosítása érdekében inkább a műveleti kódos formátumot használjuk. A ***Binutils 2.15***-ös és újabb változatai ugyanakkor, ahol kell – ilyen például a ***gas (GNU assembler)*** és az ***objdump*** –, ott ismerik a szimbolikus neveket is. A ***binutils*** többek közt az utasításszintű műveletekért felelős ***BFD*** könyvtárát használja a ***GNU gdb*** hibakereső is. Egy titkosító függvény kiíratása például így nézhet ki:

```
(gdb) x/3i $pc
0x8048392 <demo1+14>:    lea    0x80495f0,%edx
0x8048398 <demo1+20>:    repz xcryptecb
0x804839c <demo1+24>:    push   %eax
```

Mint sejteni lehetett, a ***SUSE Linux 9.2*** minden vonatkozó csomagja rendelkezik ***PadLock*** foltokkal, így a ***PadLock*** támogatás ennél a terjesztésnél már gyári állapotban is megoldott. Akinek a terjesztése nem rendelkezik a foltokkal, az látogasson el a források között szereplő ***Linux PadLock*** honlapomra, és keresse meg a számára szükségeseket.

## A PadLock programozása

Az alábbiakban néhány a ***PadLock*** programozásával kapcsolatos irányelvet szeretnék ismertetni, és kicsit részletesebben is foglalkozni fogok az `xcryptcbc`-vel. Példát mutatok a ***PadLock*** alkalmazására egy adatpuffer ***AES*** algoritmussal, 128 bites kulccsal, ***CBC*** módban végzett titkosítására. Az `xcrypt` csoport összes többi utasítása is pontosan így használható, illetve az egyéb ***PadLock***-szolgáltatások is hasonlóan működnek.

### xcryptcbc

Az `xcryptcbc` semmilyen explicit operandussal nem rendelkezik. Ehelyett minden regiszternek meghatározott szerepet ad:

- `ESI` – forráscím.
- `EDI` – célcím.
- `EAX` – kezdeti értékadási vektorcím.
- `EBX` – titkosító kulcs címe.
- `ECX` – a feldolgozandó blokkok száma.
- `EDX` – vezérlőszó címe.

Alapesetben minden címet 16 bájtos határra kell igazítani.

### ESI/EDI – A forrás- és céladatok címei

A forrás- és a célcím akár azonos is lehet, vagyis a titkosítást helyben is végezhetjük. A célpuffernek legalább akkorának kell lennie, mint a forrásnak. Mindkettő méretének a blokkméret (16 bájt) többszörösének kell lennie. Egyes esetekben az ***Esther*** processzor az igazítás nélküli pufferek használatát is lehetővé teszi, de ilyenkor a művelet végrehajtása lassabb.

### EAX – kezdeti értékadási vektorcím

A ***kezdeti értékadási vektor (initialization vector, IV)*** egyike a titkosítás eredményét meghatározó átadott értékeknek. Az ***IV*** mérete megegyezik a blokkmérettel, vagyis 16 bájt. A kezdeti értékadási vektorokról a megfelelő szakirodalomban lehet részletesebben olvasni.

### EBX – titkosító kulcs címe

A titkosító kulcs mérete 128, 192 vagy 256 bit lehet. Az ***AES*** algoritmus belül úgynevezett kiterjesztett kulcsot használ, amit a titkosító kulcsból származtat. A 128 bites kulcsoknál a kiterjesztett kulcsot a ***PadLock*** számítja, a hosszabb kulcsoknál ezt nekünk kell megtennünk.

### ECX – feldolgozandó blokkok száma

Az `xcrypt` utasításokat mindig a `rep` előtaggal használjuk, ami lehetővé teszi ismételt végrehajtásukat egészen addig, amíg az `ECX` regiszter értéke nulla nem lesz. Az `ECX` által tárolt érték az egyes blokkok titkosításakor vagy visszafejtésekor fokozatosan csökken.

### EDX – vezérlőszó címe

A ***PadLock*** csak akkor tudja, hogy pontosan hogyan kell feldolgoznia az adatokat, ha az úgynevezett vezérlőszó adatszerkezetet feltöltjük a következő elemekkel:

- Algoritmus (`algo`) – csak az ***AES***-t választhatjuk.
- Kulcsméret (`ksize`) – a támogatott méretek egyike.
- Irány (`encdec`) – titkosítás vagy visszafejtés.
- Kulcselőállítás (`keygen`) – elkészítettük a kiterjesztett kulcsot, vagy a ***PadLocknak*** kell kiszámítania?
- Körök (`rounds`) – az algoritmus egyik belső értéke, jelentését lásd a későbbiekben vagy a ***PadLock*** leírásában.

C-ben egy `union` tökéletesen megfelel az adatszerkezet helyének lefoglalására, elemeit pedig egy bitmező segítségével könnyen meg tudjuk adni és el tudjuk érni:

```
union cword {
    uint8_t cword[16];
    struct {
        int rounds:4;
        int algo:3;
        int keygen:1;
        int interm:1;
        int encdec:1;
        int ksize:2;
    } b;
};
```

## Assembler példa

Elméletből mindent tudunk, nézzünk egy valódi példát. Kezdjünk néhány tiszta assembler-sorral:

```
.comm   iv,16,16
.comm   key,16,16
.comm   data,16,16
.comm   cword,16,16
.text
cryptcbc:
    movl    $data, %esi  #; forráscím
    movl    %esi, %edi   #; cél
    movl    $iv, %eax    #; IV
    movl    $key, %ebx   #; titkosító kulcs
    movl    $cword, %edx #; vezérlőszó
    movl    $1, %ecx     #; blokkszám
    rep xcryptcbc
    ret
```

A fenti kódrészlet megadott titkosító kulccsal és kezdeti értékadási vektorral, a `cword` vezérlőszóban megadottak szerint titkosít egy adatblokkot. Megjegyzem, hogy mivel helyben titkosítunk, a forrás- és a célcím azonos. Mivel a `data` mező mérete egyetlen blokknyi, az `ECX` regiszter értékeként egyet adtunk meg.

## C példa

Ha a ***PadLockot*** közvetlenül, ***C*** programból akarjuk használni, akkor megtehetjük például, hogy a ***PadLock*** eljárásokat külön assembler forrásfájlokba helyezzük, majd a fordítást különálló modulokba végezzük, végül összekapcsoljuk a bináris fájlt. Egyszerűbb azonban a ***GCC*** beépített assemblerét használni, és az utasításokat közvetlenül a ***C*** kódba illeszteni. A források között szerepel egy a beépített assemblerrel kapcsolatos oktatóanyagra mutató hivatkozás.

```
static inline void *
padlock_xcryptcbc(char *input, char *output,
    void *key, void *iv, void *control_word,
    int count)
{
    asm volatile ("xcryptcbc"
        : "+S"(input), "+D"(output), "+a"(iv)
        : "c"(count), "d"(control_word),
        ➥"b"(key));
    return iv;
}
```

A fenti kódrészlet a fordítót a megfelelő bemeneti, számláló és egyéb átadott értékek megfelelő regiszterekbe való betöltésére utasítja. Ezután sor kerül az `xcryptcbc` utasítás kiadására, majd az EAX regiszterben talált értékkel térünk vissza, mint az új kezdeti értékadási vektorra irányuló mutatóval. Ügyeljünk a vezérlőszó adatszerkezet helyes kitöltésére. A legjobb az, ha első lépésként töröljük az `union`-t, így elkerülhetjük a memóriában esetlegesen előforduló értékek használatát:

```
memset(&cword, 0, sizeof(cword));
```

Most egyenként töltsük fel a mezőket. A lista első eleme a `rounds`. Ez adja meg, hogy az ***AES*** algoritmus hányszor menjen végig a bemeneti blokkon. Az algoritmus minden körben a kiterjesztett kulcs egy egyedi részét használja. Ha a ***FIPS AES*** szabványát akarjuk követni, akkor 128 bites kulcsnál 10, 192 bitesnél 12, 256 bitesnél pedig 14 kört kérjünk. Ha a `key_size` változó a titkosító kulcs méretét bájtban adná meg, akkor a körök számát az alábbi képlettel kapnánk:

```
cword.b.rounds = 10 + (key_size - 16) / 4;
```

A következő mező az `algo`. Segítségével a későbbiekben más, az ***AES***-től eltérő algoritmusokat is választhatunk majd, ám jelenleg kizárólag az ***AES*** áll rendelkezésünkre. Értékét tehát hagyjuk nullán.
A `keygen` mezőt akkor állítsuk egyre, ha a kiterjesztett kulcsot magunk állítjuk elő. A nullás érték azt jelenti, hogy a ***PadLocknak*** magának kell azt előállítania; erre viszont csak 128 bites kulcsnál van lehetőség:

```
cword.b.keygen = (key_size > 16);
```

Az `interm` lehetővé teszi az algoritmus futtatásának egyes körei után kapott köztes értékek tárolását. Van egy olyan gyanúm, hogy a processzor tervezői ezt a mezőt a processzormag hibáinak felderítésére használták, alkalmazásokon belüli használatának nem látom értelmét.
A titkosítás és a visszafejtés megkülönböztetése az `encdec` bittel történik. A nulla titkosítást, az egy visszafejtést jelent. Végül a kulcsméretet a `ksize` két bittel adja meg:

```
cword.b.ksize = (key_size - 16) / 8;
```

Ennyi. A vezérlőszó előkészítése és a pufferek igazítása után meghívhatjuk a `padlock_xcryptcbc()`-t. Ha a gépben keringő elektronok is úgy akarják, néhány pillanat múlva megkapjuk a titkosított adatokat.

## Összefoglalás

A ***PadLock*** leírása nyilvánosan elérhető a ***VIA*** webhelyén, ahol további a ***PadLock*** programozásával kapcsolatos anyagokat is találni. Saját honlapomon szerepel egy teljes, a ***PadLock*** segítségével egyetlen adatblokkot titkosító példaprogram. A források között további figyelemre érdemes hivatkozások is szerepelnek.

*Linux Journal 2005. május, 133. szám*

A cikk forrásai: ➲ ***www.linuxjournal.com/article/8137***

***Michal Ludvig*** (michal@logix.cz) nemrég költözött Prágából a világ másik felére, Aucklandbe, ahol az Asterisk Ltd. vezető mérnöke. Feleségével és lányával mostanában Új-Zéland titkainak felfedezésével töltik szabadidejüket.