Brute-Force Attacks on Whole-Disk Encryption

Attacking Passwords Instead of Encryption Keys

Gregory Hildstrom, CISSP
Product Development
Raytheon Trusted Computer Solutions
San Antonio, TX, USA

November, 18 2012

***Abstract*—This paper examines some simple brute-force methods of password recovery for dm-crypt encrypted hard disk drives. The methods are described, performance is analyzed, and the attack method is compared to brute-force attacking the encryption key instead of the password.**

Keywords—dm-crypt; encryption; ESSIV; attack; AES

I. INTRODUCTION

Dm-crypt, which is included in most recent Linux kernels, provides "transparent encryption of block devices using the kernel crypto API". [1] It is built on the same underlying device-mapper infrastructure the Linux kernel already uses to support redundant arrays of inexpensive disks (RAID) and logical volume management (LVM). Dm-crypt has access to ciphers, hashes, and block modes supported by the running kernel and whatever crypto API modules are currently loaded. The default full specification is aes-cbc-essiv:sha256, which indicates AES with a 256-bit key for the cipher, cipher block chaining (CBC) for the mode, encrypted sector-salt initialization vector (ESSIV) for the per-sector initialization vector, and SHA256 for hashing. Because this is the default for dmsetup and cryptsetup when no additional command-line options are specified, most users of Linux disk encryption probably use these default settings. [1][2] This paper focuses its efforts on brute force password recovery of dm-crypt devices using this specification.

II. SEARCH SPACE

Attempting to recover a typical password involves a much smaller search space than attempting to recover the encryption key. For example, a 256-bit key has 2^256 = 1.16E+77 possible combinations and a strong 8-character password has 94^8 = 6.10E+15 possible combinations. The password search space is at least 1.0E+62 times smaller than the encryption key search space. There is a huge computational benefit even if checking each password is 1000 times more computationally intensive than checking each encryption key.

III. TEST SYSTEM

My test system was a dual-core 64-bit HP Pavilion g-series laptop with 4GB RAM running Microsoft Windows 7 64-bit. The Linux work was performed on a VMware Workstation 8 virtual machine with one processor core and 1GB RAM running CentOS 6 64-bit.

IV. BASH, DM-CRYPT, DIFF TESTING

I started by creating a proof-of-concept bash script that created an encrypted dm-crypt device, initialized it with some known plaintext, and recovered the password by testing every possible combination using operating system tools. For my testing, I used 4-digit numeric passwords, like "1234", to prove the concepts in a reasonable amount of time. My dmcrypt-known.sh script performed the following steps:

- Create empty file
- Add file as a loopback device with losetup -a
- Create the dm-crypt device with cryptsetup create
- Initialize the dm-crypt device with known plaintext
- Remove the dm-crypt device with cryptsetup remove
- Loop possible passwords
    - Create the dm-crypt device with cryptsetup create
    - Test known plaintext with dd/diff
    - Print the password and exit if found
    - Remove the dm-crypt device with cryptsetup remove

V. BASH, DM-CRYPT, MOUNT TESTING

Next I created another bash script that created an encrypted dm-crypt device, initialized it with an ext3 file system, and again recovered the password by testing every possible combination using operating system tools. My dmcrypt-mount.sh script performed the following steps:

- Create empty file
- Add file as a loopback device with losetup -a

- Create the dm-crypt device with cryptsetup create
- Create a file system on the dm-crypt device with mkfs.ext3
- Remove the dm-crypt device with cryptsetup remove
- Loop possible passwords
    - Create the dm-crypt device with cryptsetup create
    - Try to mount the dm-crypt device with mount
    - Print the password and exit if found
    - Remove the dm-crypt device with cryptsetup remove

VI. Cipher Block Mode

The timing results for these first two approaches led me to wonder if the overhead of setting up these transparent devices in the kernel was significant compared to the actual hashing and encryption operations, which led me to dig a bit deeper into how this default specification actually works. The smallest amount of data that the operating system can read from or write to a hard disk drive is one sector, which is 512 bytes. To prevent undesirable cascading effects, each sector is encrypted and decrypted independently as shown in Figure **1**. Ciphers like AES only encrypt or decrypt 16 bytes, 128 bits, at a time. Thirty two of these cipher blocks are required to fill one sector. If each cipher block were encrypted independently with no other alterations, many attacks would be possible that reduce the strength of the encryption scheme below brute-force strength. To counteract this, cipher blocks are combined with each other after and before encryption in various cipher-block modes. Similarly, if sectors are treated independently with no alterations, other attacks would be possible that reduce the strength of the encryption scheme. To counteract this, each sector is modified or initialized with some data that makes the ciphertext too difficult to analyze. The cipher block chaining (CBC) mode combines the previous encrypted block with the current plaintext block prior to encryption of the current block as shown in Figure **2**. [3]

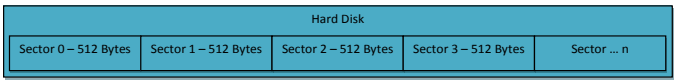

Figure 1. Hard Disk Sectors

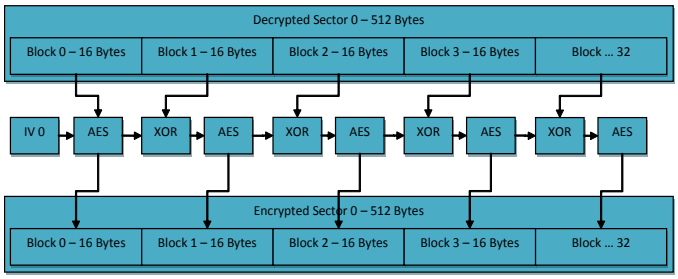

Figure 2. Cipher Block Chaining (CBC) Mode

VII. Encrypted Salt-Sector Initialization Vector (ESSIV)

The strength of CBC is greatly improved if the initialization data is difficult for an attacker to determine. Many simple schemes used the sector number as the initialization data, but that is very easy for an attacker to determine. As a result, Fruhwirth developed the encrypted salt-sector initialization vector (ESSIV) approach. The ESSIV approach encrypts the sector number with a hash of the encryption key to determine the IV. This makes the IV as difficult for an adversary to guess as the encryption key itself because it combines key data with a sector number that changes for each sector. [3]

VIII. Bash, openssl, diff Testing

Next I created another bash script that replicated dm-crypt kernel operations with openssl to eliminate the entire kernel device overhead. My openssl-kown.sh script performed the following steps:

- Derive the key from the password with sha256sum
- Derive the salt from the key with sha256sum
- Derive the IV from the salt and the sector number with openssl
- Encrypt the known plaintext sector and save it as a new sector with openssl
- Loop possible passwords
    - Derive the key from the password with sha256sum
    - Derive the salt from the key with sha256sum
    - Derive the IV from the salt and the sector number with openssl
    - Decrypt the encrypted sector with openssl
    - Compare it to the known plaintext with diff
    - Print the password and exit if found

IX. Performance Results

The openssl approach was about 2.5 times faster than using dm-crypt in the kernel because of the lower overhead. I imagine that the process could be sped up by at least a factor of 10 if a custom C program using the openssl libraries were created. A custom program would perform faster than interpreted bash scripting and it would eliminate many system calls like forking other processes and file system I/O compared to my test script. In addition, a cluster with 1600000 processor cores or 200000 8-core processors would be orders of magnitude faster. Table **1** shows some actual timings and performance estimates.

Table 1. Performance Results

brute force implementation	seconds	passwords/second
bash/cryptsetup/diff	270	4.57
bash/cryptsetup/mount	250	4.94
bash/openssl/diff	100	12.34
c/estimated	10	123.40
c/cluster/estimated	6.25E-06	1.97E+08

X. Feasibility and Estimates

The next question relates to feasibility of this approach. Cracking a 4-digit personal identification number (PIN) style password proved easy, but an 8-character strong (not dictionary) password is much more difficult. My simple openssl-known.sh script would require 1.57E+07 years to test every possible 8-character strong password, which is not feasible. A 10 times faster C program would require 1.57E+06 years to test every possible 8-character strong password, which is still not feasible. The 200000-node 8-core cluster would require just under a year to test every possible 8-character strong password, but that computer would be near the top of the world's most powerful supercomputers list. The problem becomes orders of magnitude more difficult if the password is 9, 10, 11, or 12 characters using the entire keyboard character set. Interestingly, a password would need to be 40 characters long with a 94-character set to equal the key space of the 256-bit encryption key. Table **2** shows estimated brute force times for various search spaces.

Table 2. Estimates

digits	4	8	39	40	256
value set	0-9	a-zA-Z0-9~-/	a-zA-Z0-9~-/	a-zA-Z0-9~-/	0-1
values	10	94	94	94	2
combinations	1.00E+04	6.10E+15	8.95E+76	8.42E+78	1.16E+77
bash/openssl/diff years	2.57E-05	1.57E+07	2.30E+68	2.16E+70	2.97E+68
c/estimated years	2.57E-06	1.57E+06	2.30E+67	2.16E+69	2.97E+67
c/cluster/estimated years	1.60E-12	9.78E-01	1.44E+61	1.35E+63	1.86E+61

XI. Conclusions

The brute force approach described in the paper could be made much more powerful if a more intelligent password-generating program like John The Ripper (JTR) were used. The dictionary-based approach and word-mangling algorithms in JTR have proven effective in password recovery from hashes and it can fall back to pure brute force when necessary. This same strategy would most likely be effective against TrueCrypt and PGP Whole Disk encryption even though the algorithm details will vary slightly. Any time an encryption key is derived from the hash of a password and the password length and password complexity result in significantly fewer combinations than the binary encryption key, it is better to directly attack the password than the key.

XII. Bash Scripts

A. dmcrypt-known.sh

```
#!/bin/bash

### variables ###
SECTORSIZE=512
SECTORS=100000
INITFILE="/dev/zero"
KNOWN="knownplaintext"
EFILE="encryptedfile"
EDEV="/dev/loop0"
DFILE="decryptedfile"
DDEV="/dev/mapper/$DFILE"

### create and init the empty encrypted file ###
dd bs=$SECTORSIZE count=$SECTORS if=$INITFILE
of=$EFILE
losetup $EDEV $EFILE
md5sum $EDEV $EFILE

### create encrypted file ###
echo "1234" | cryptsetup create $DFILE $EDEV
cryptsetup status $DFILE
dd bs=$SECTORSIZE if=$DDEV of=$DFILE

### init the decrypted file for known plaintext ###
md5sum $DDEV
dd bs=$SECTORSIZE if=$INITFILE of=$DDEV
md5sum $DDEV
md5sum $EFILE

### record known plaintext ###
dd bs=32 count=1 if=$DDEV of=$KNOWN

### remove the decrypted file ###
cryptsetup remove $DDEV

### password-cracking-type attack ###
for ((D3=0;$D3<=9;D3=$D3+1)); do
for ((D2=0;$D2<=9;D2=$D2+1)); do
for ((D1=0;$D1<=9;D1=$D1+1)); do
for ((D0=0;$D0<=9;D0=$D0+1)); do
        echo "Trying $D3$D2$D1$D0"
        echo "$D3$D2$D1$D0" | cryptsetup create $DFILE
$EDEV
        dd bs=32 count=1 if=$DDEV 2>/dev/null | diff -
$KNOWN >/dev/null 2>&1
        RC=$?
```

```
        if [ $RC == 0 ]; then
                echo "Password Found! $D3$D2$D1$D0"
                exit 0
        fi
        cryptsetup remove $DDEV
done
done
done
done

### clean up ###
cryptsetup remove $DFILE
losetup -d $EDEV
```

B. dmcrypt-mount.sh

```
#!/bin/bash

### variables ###
SECTORSIZE=512
SECTORS=100000
INITFILE="/dev/zero"
KNOWN="knownplaintext"
EFILE="encryptedfile"
EDEV="/dev/loop0"
DFILE="decryptedfile"
DDEV="/dev/mapper/$DFILE"

### create and init the empty encrypted file ###
dd bs=$SECTORSIZE count=$SECTORS if=$INITFILE
of=$EFILE
losetup $EDEV $EFILE
md5sum $EDEV $EFILE

### create encrypted file ###
echo "1234" | cryptsetup create $DFILE $EDEV
cryptsetup status $DFILE
mkfs.ext3 $DDEV

### remove the decrypted file ###
cryptsetup remove $DDEV

### password-cracking-type attack ###
for ((D3=0;$D3<=9;D3=$D3+1)); do
for ((D2=0;$D2<=9;D2=$D2+1)); do
for ((D1=0;$D1<=9;D1=$D1+1)); do
for ((D0=0;$D0<=9;D0=$D0+1)); do
        echo "Trying $D3$D2$D1$D0"
        echo "$D3$D2$D1$D0" | cryptsetup create $DFILE
$EDEV
        mount $DDEV /mnt/dmcrypt >/dev/null 2>&1
        RC=$?
        if [ $RC == 0 ]; then
                echo "Password Found! $D3$D2$D1$D0"
                exit 0
        fi
        umount $DDEV >/dev/null 2>&1
        cryptsetup remove $DDEV
done
done
done
done

### clean up ###
cryptsetup remove $DFILE
losetup -d $EDEV
```

C. openssl-known.sh

```
#!/bin/bash

encrypt()
{
PASSPHRASE=$1
PLAINTEXTFILE=$2
ENCRYPTEDFILE=$3
SECTOR="0"

### create the encryption key from passphrase ###
KEYHEX=`echo -n $PASSPHRASE | sha256sum | awk
'{print $1}' | tr -d '\n'`
#echo KEYHEX: $KEYHEX

### create the salt from the key ###
SALTHEX=`echo -n $KEYHEX | sha256sum | awk '{print
$1}' | tr -d '\n'`
#echo SALTHEX: $SALTHEX

### create the iv from the salt ###
IVHEX=`echo -n 0 | openssl enc -aes-256-cbc -e -K
$SALTHEX -iv 0 | md5sum | awk '{print $1}' | tr -d '\n'`
#echo IVHEX: $IVHEX

### create encrypted file ###
openssl enc -aes-256-cbc -in $PLAINTEXTFILE -out
$ENCRYPTEDFILE -e -K $KEYHEX -iv $IVHEX
}

decrypt()
{
PASSPHRASE=$1
ENCRYPTEDFILE=$2
PLAINTEXTFILE=$3
SECTOR="0"

### create the encryption key from passphrase ###
KEYHEX=`echo -n $PASSPHRASE | sha256sum | awk
'{print $1}' | tr -d '\n'`
#echo KEYHEX: $KEYHEX

### create the salt from the key ###
SALTHEX=`echo -n $KEYHEX | sha256sum | awk '{print
$1}' | tr -d '\n'`
#echo SALTHEX: $SALTHEX

### create the iv from the salt ###
IVHEX=`echo -n 0 | openssl enc -aes-256-cbc -e -K
$SALTHEX -iv 0 | md5sum | awk '{print $1}' | tr -d '\n'`
```

```
#echo IVHEX: $IVHEX

### create encrypted file ###
openssl enc -aes-256-cbc -in $ENCRYPTEDFILE -out
$PLAINTEXTFILE -d -K $KEYHEX -iv $IVHEX >/dev/null
2>&1

return $?
}

### variables ###
SECTORSIZE=512
SECTORS=1
INITFILE="/dev/zero"
KNOWN="knownplaintext"
EFILE="encryptedfile"
EDEV="/dev/loop0"
DFILE="decryptedfile"
DDEV="/dev/mapper/$DFILE"
KEYHEX=""
SALTHEX=""
IVHEX=""

### create known sector file ###
dd bs=$SECTORSIZE count=$SECTORS if=$INITFILE
of=$KNOWN

### create encrypted file ###
encrypt "1234" $KNOWN $EFILE

### password-cracking-type attack ###
for ((D3=0;$D3<=9;D3=$D3+1)); do
for ((D2=0;$D2<=9;D2=$D2+1)); do
for ((D1=0;$D1<=9;D1=$D1+1)); do
for ((D0=0;$D0<=9;D0=$D0+1)); do
        echo "Trying $D3$D2$D1$D0"
        decrypt "$D3$D2$D1$D0" $EFILE $DFILE
        diff $DFILE $KNOWN >/dev/null 2>&1
        RC=$?
        if [ $RC == 0 ]; then
                echo "Password Found! $D3$D2$D1$D0"
                exit 0
        fi
done
done
done
done

### clean up ###
```

References

[1] Broz, M. (2012). *dm-crypt: Linux kernel device-mapper crypto target*. Retrieved November 18, 2012, from http://code.google.com/p/cryptsetup/wiki/DMCrypt

[2] Red Hat, Inc. (2010). *cryptsetup(8) - Linux man page*. Retrieved November 18, 2012, from http://linux.die.net/man/8/cryptsetup

[3] Fruhwirth, C. (2005). *New Methods in Hard Disk Encryption*. Retrieved November 18, 2012, from http://clemens.endorphin.org/nmihde/nmihde-A4-os.pdf