A dive into the world of MS-DOS viruses

blog.benjojo.co.uk/post/dive-into-the-world-of-dos-viruses

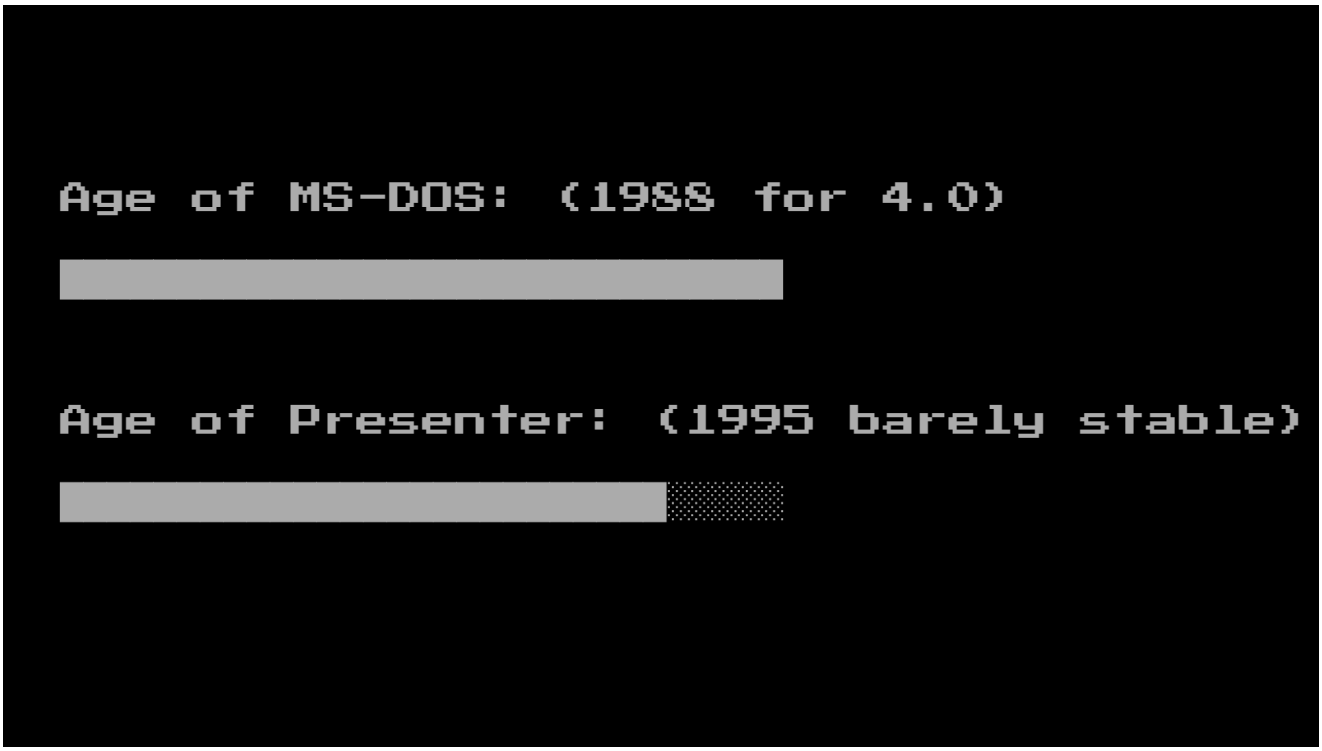
Jan 4 2019

Translations are available in: русский

This post is a textual version of a talk I gave at The 35th Chaos Computer Congress at the end of 2018. You can watch the talk that was recorded by the wonderful C3VOC team below if that's your preferred medium:

Watch Video At: https://youtu.be/xgS1M4e_9_E

Or watch using the C3VOC/media.ccc.de player



So I have an admission to make, MS-DOS does slightly outage me, regardless MS-DOS malware has always fascinated me to some degree, but first we must ask: “What is DOS?”

- DOS is the “one up” of CP/M, another very old operating system
- The DOS family covers a wide range of vendors, just because it’s DOS does not mean it’s going to be running on a 8086 CPU or better
- Some of these DOS vendors share API compatibility, meaning that some have shared malware!

But really, most of our memories of the DOS era is strong aesthetic for how the computers of the looked at the time:

This is the era of “computing beige” and the Model M keyboard, that may be famous or infamous depending on if you enjoy loud keyboards or not.

```
HIMEM is testing extended memory...done.


This driver is provided by Oak Technology, Inc..
OTI-91X ATAPI CD-ROM device driver, Rev D91XV352
(C)Copyright Oak Technology Inc. 1987-1997
  Device Name         : 12345678
  Transfer Mode       : Programmed I/O
  Number of drives    : 1



C:\>C:\DOS\SMARTDRV.EXE /X
MSCDEX Version 2.23
Copyright (C) Microsoft Corp. 1986-1993. All rights reserved.
       Drive E: = Driver 12345678 unit 0

CuteMouse v1.9 [FreeDOS]
Installed at PS/2 port
The command completed successfully.
C:\>D:

D:\>w
```

Some of us may have memories of using DOS, and some might still use DOS!

For example, George R R Martin who wrote Game of Thrones reportedly uses Wordstar on DOS to write the book!

We also cannot overlook QBASIC, for many this would have been their first exposure to programming!

```
A:\>GOTO :1

A:\>WORK.COM
Hello - This is a 100   COM test file, 1993

A:\>GOTO :1

A:\>WORK.COM
Hello - This is a 100   COM test file, 1993

A:\>GOTO :1

A:\>WORK.COM
Hello - This is a 100   COM test file, 1993

A:\>GOTO :1

A:\>WORK.COM
Hello - This is a 100   COM test file, 1993

A:\>GOTO :1

A:\>WORK.COM
Hello - This is a 100   COM test file, 1993
_
```

But sometimes life using DOS was not so great, sometimes you would be using DOS and all of a sudden things like this would happen. This sample also plays a small tune on the PC speaker while it's printing, so this could be really embarrassing in a office environment.

Some are a little more "cute", this example just shows a ascii art ambulance scrolling across the screen, and then allows the program you ran to continue, at worst a mild inconvenience.

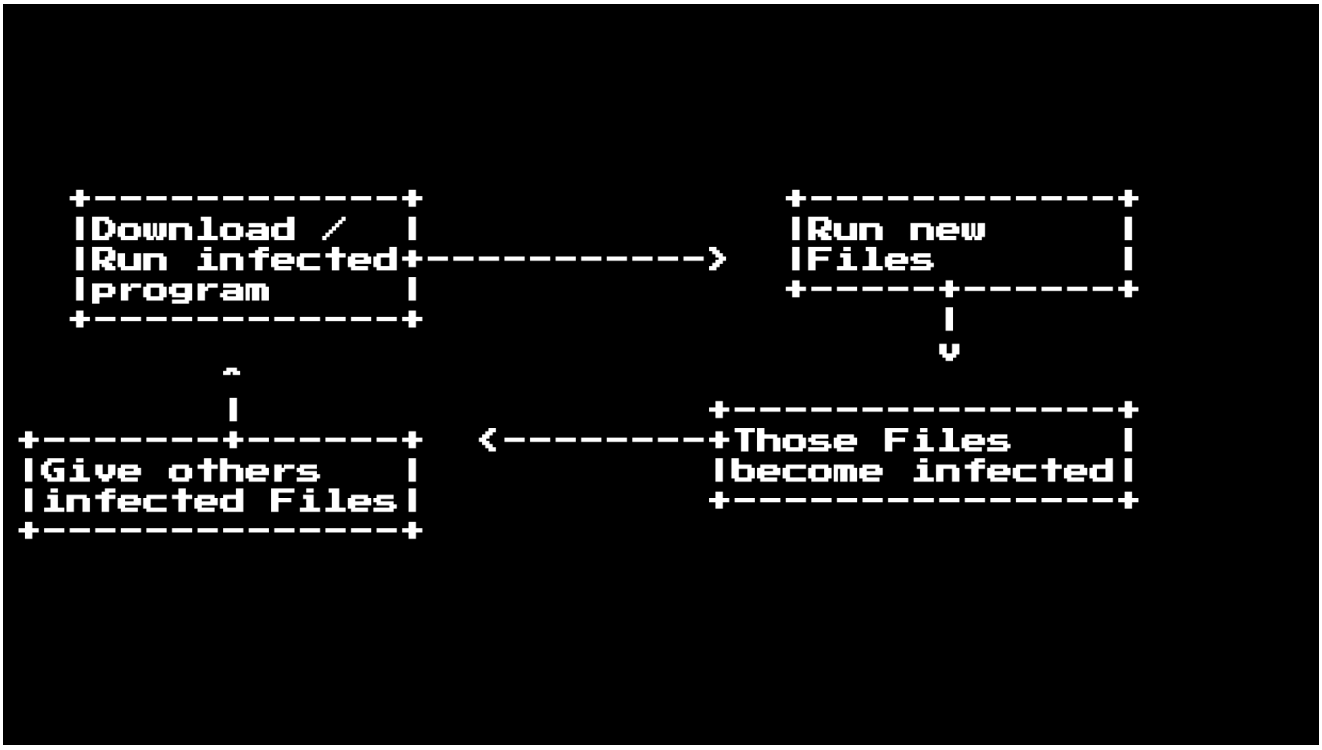
Thanks to a bunch of archivists for malware running under the name VX Heavens, we have a good historical archive of DOS Malware, or at least we would until the Ukrainian Police would raid the site:

> Friday, 23 March, the server has being seized by the police forces due to the criminal investigation (article 361-1 Criminal Code of Ukraine - the creation of the malicious programs with an intent to sell or spread them) based on someone's tip-off on "placement into the free access malicious software designed for the unauthorized breaking into computers, automated systems, computer networks".

Luckily, there are still copies of the sites database around on *popular torrent websites* that can provide us a lovely dataset:

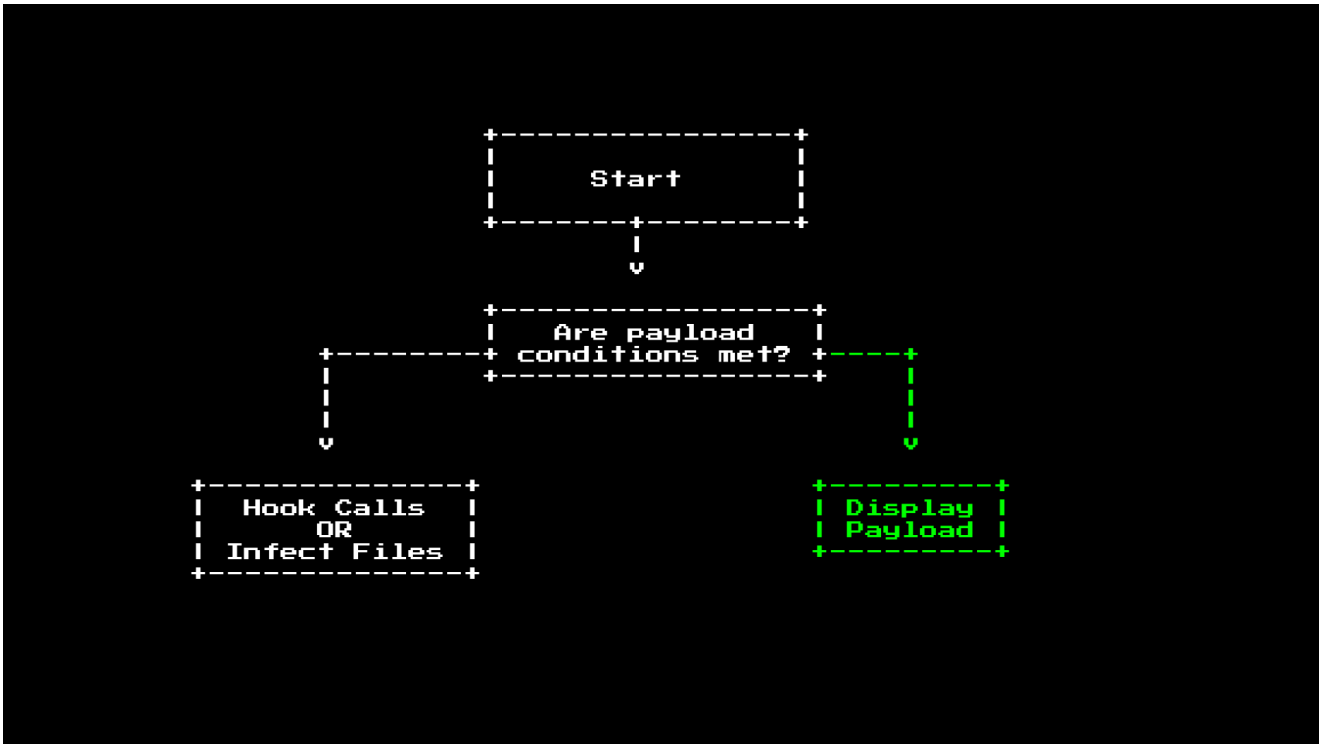
```
$ tar -tvf viruses-20070914.tar | wc -l
66714

$ ls -alh viruses-20070914.tar
6.6G viruses-20070914.tar
```

However to begin to take a look into these samples, we need to at first understand the typical propagation flow of these samples, giving that these programs are running in a pre-internet era:



Once you have got an infected file on your system and run it, the malware will either actively search or install syscall hooks to programs you run after. It will often do this in a subtle and non visible way to avoid detection. The importance of subtlety is important since to spread

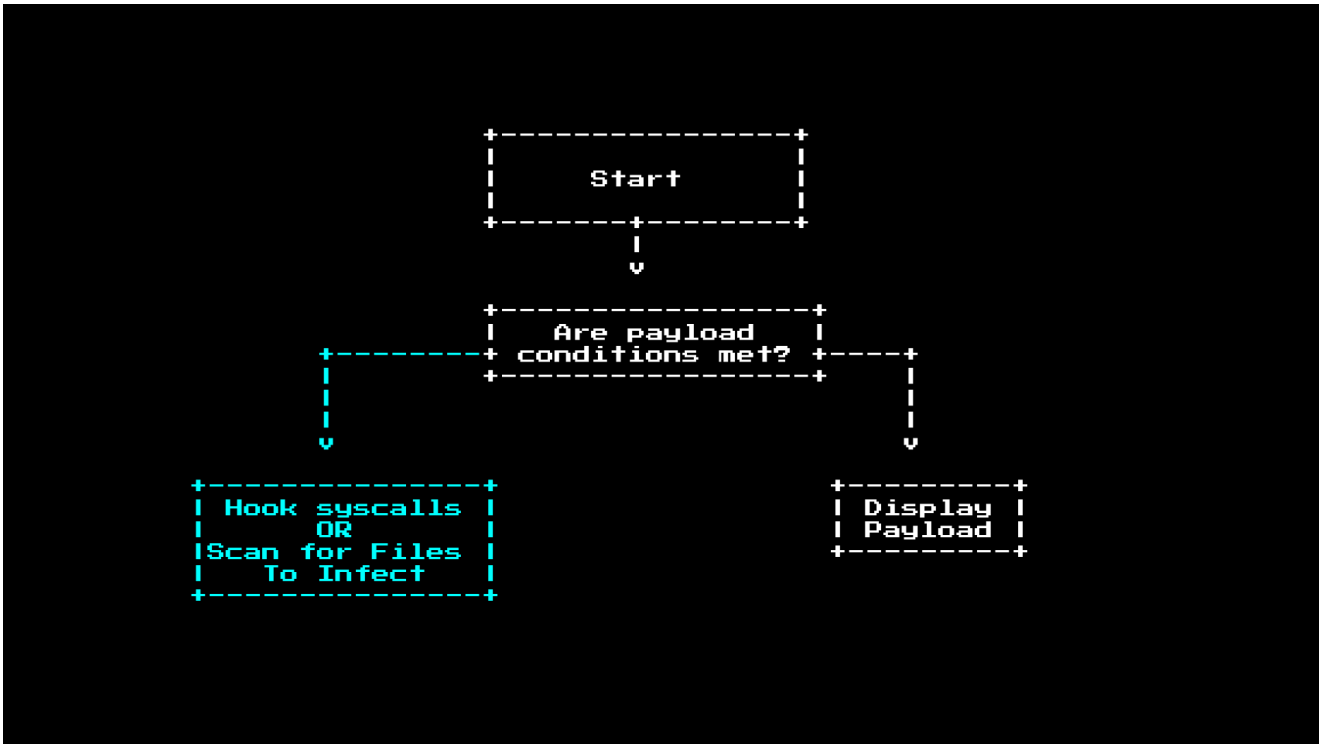
this malware need to either be given to another system through media (floppy disk) copy, or uploaded to another distribution point like a BBS



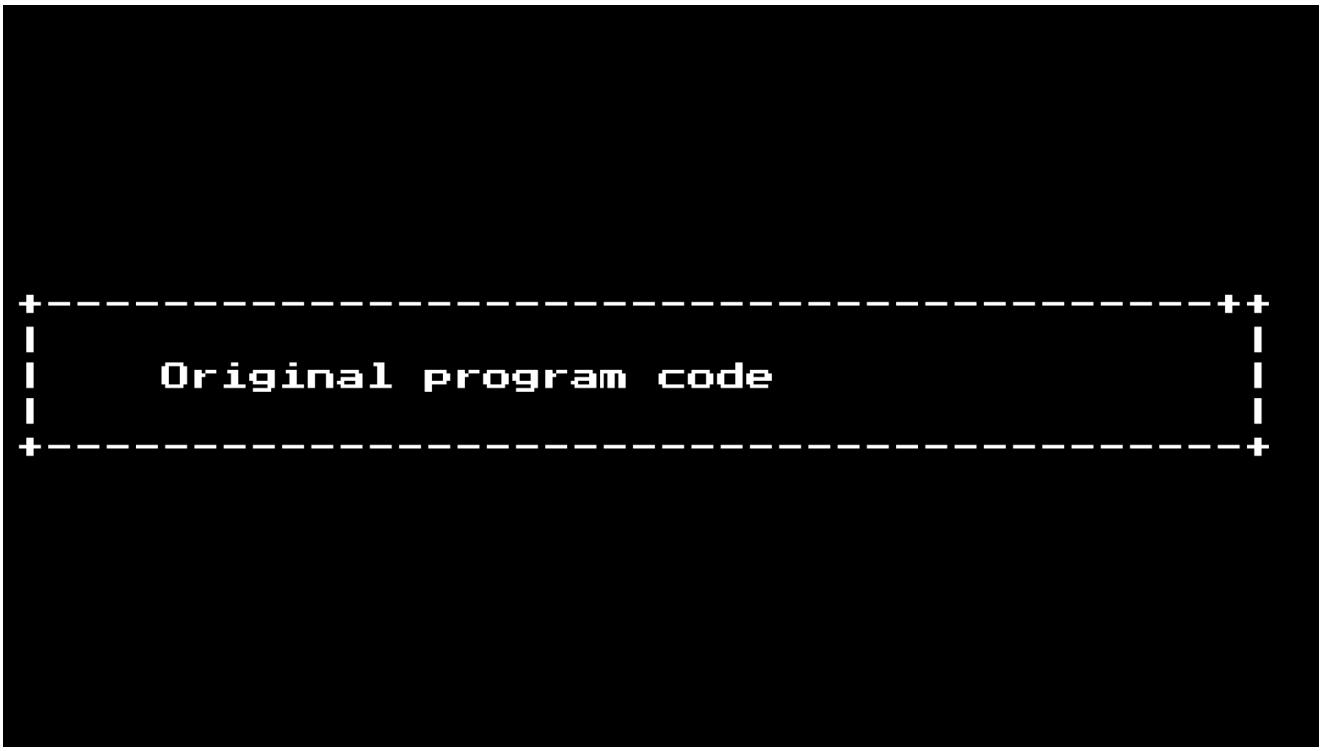
At runtime, the malware has two options; it can either stay hidden and infect new files, or it can display it's payload.

Some of the payloads are quite pretty! With the below example using fancy features such a 256 color:

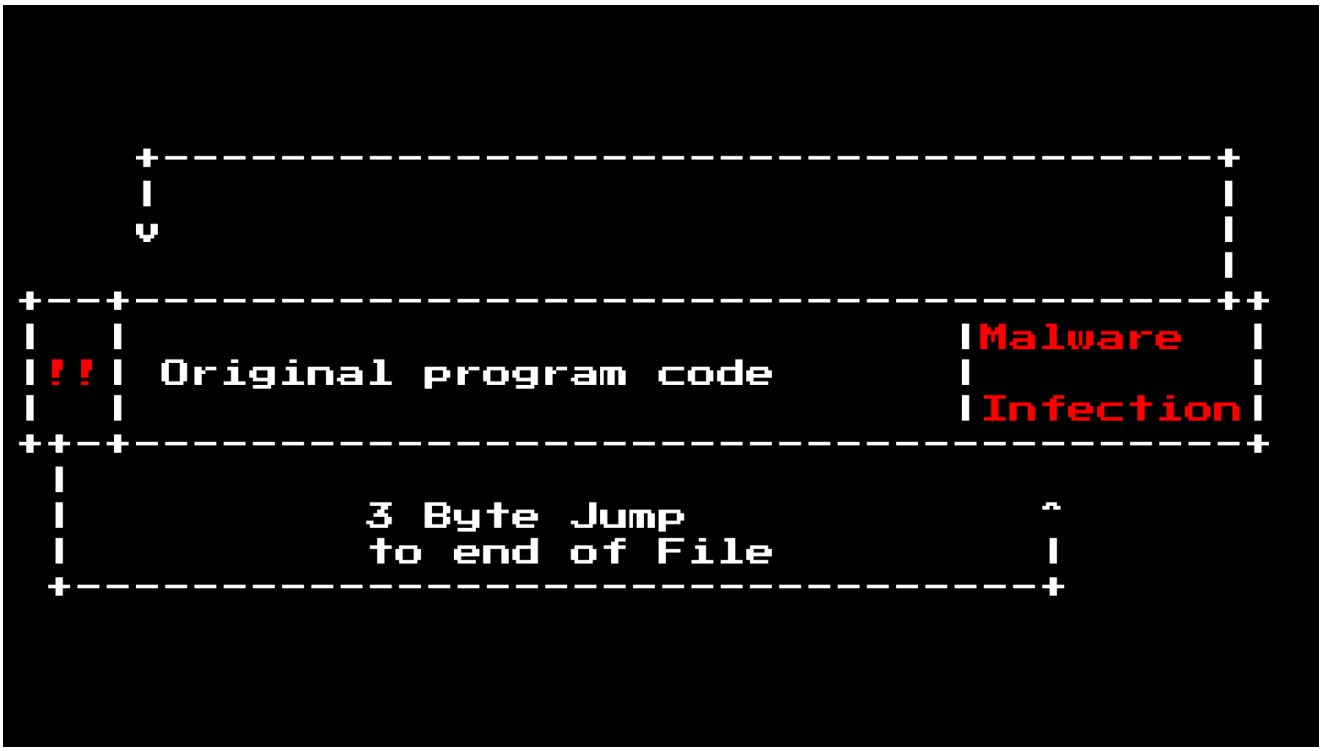
Or this one that is playing around with your screen buffer:



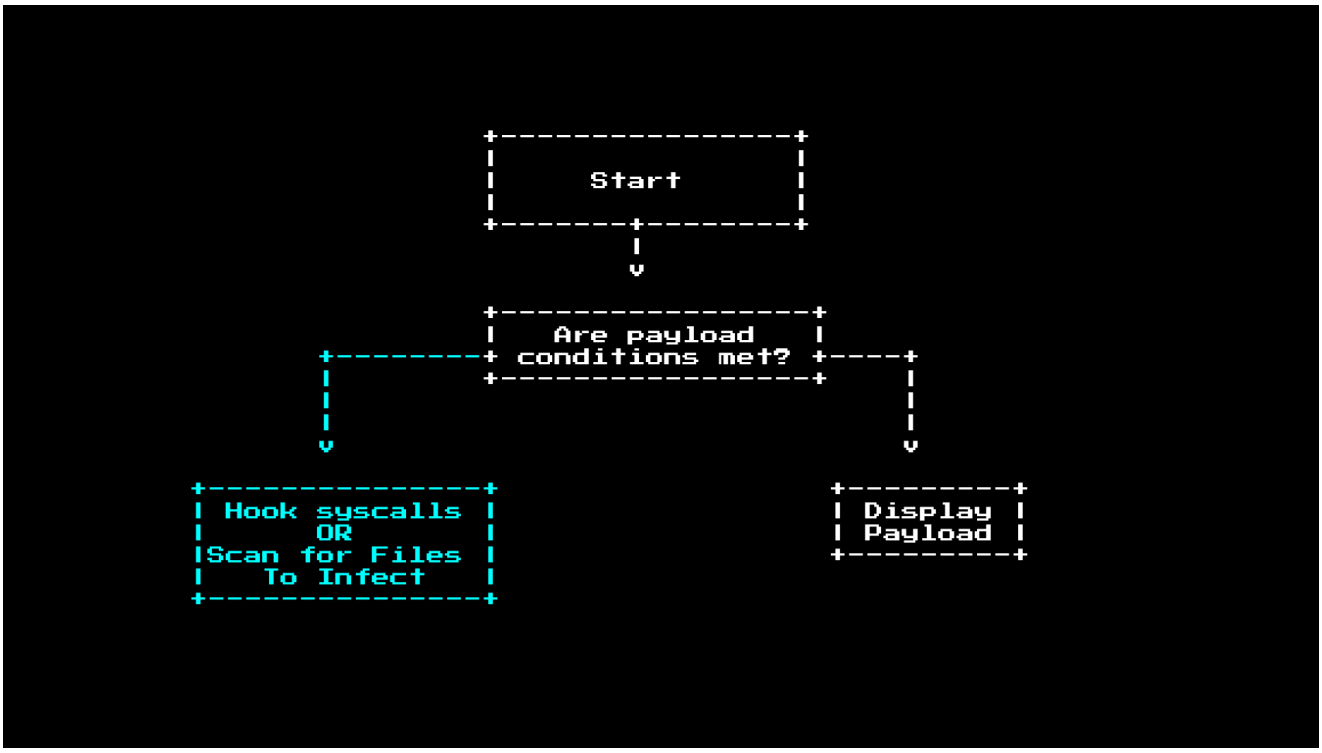
However for the most part the malware will stay quiet and try and find files to infect. Infection of most files are super easy, for example, if you view a COM file as a long tape of machine code:



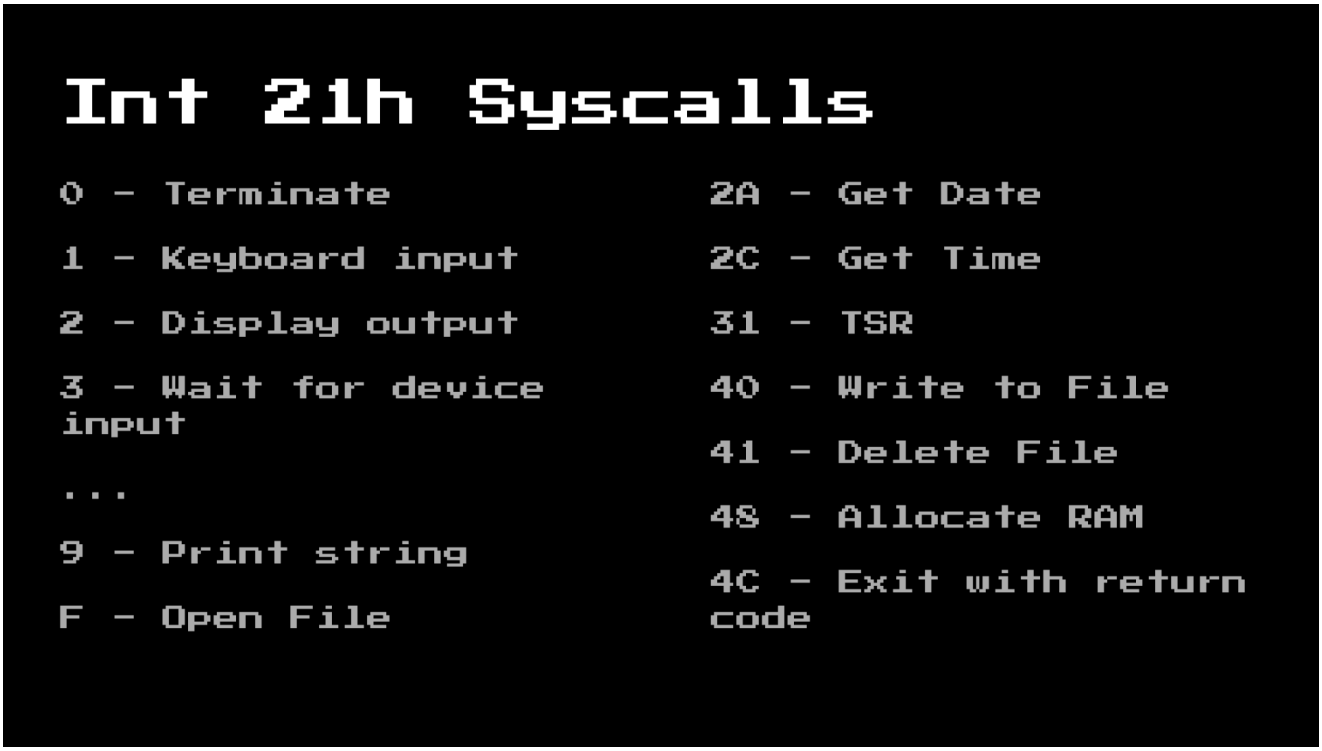
Then “all you need to do” is insert a JMP at the start of the program, and append the data to the end of the program. Leaving you with something that looks like this:



Some code was smarter and would find “empty space” in a binary and rewrite itself there, this prevented a binary from getting bigger, a possible red flag for a antivirus to use.

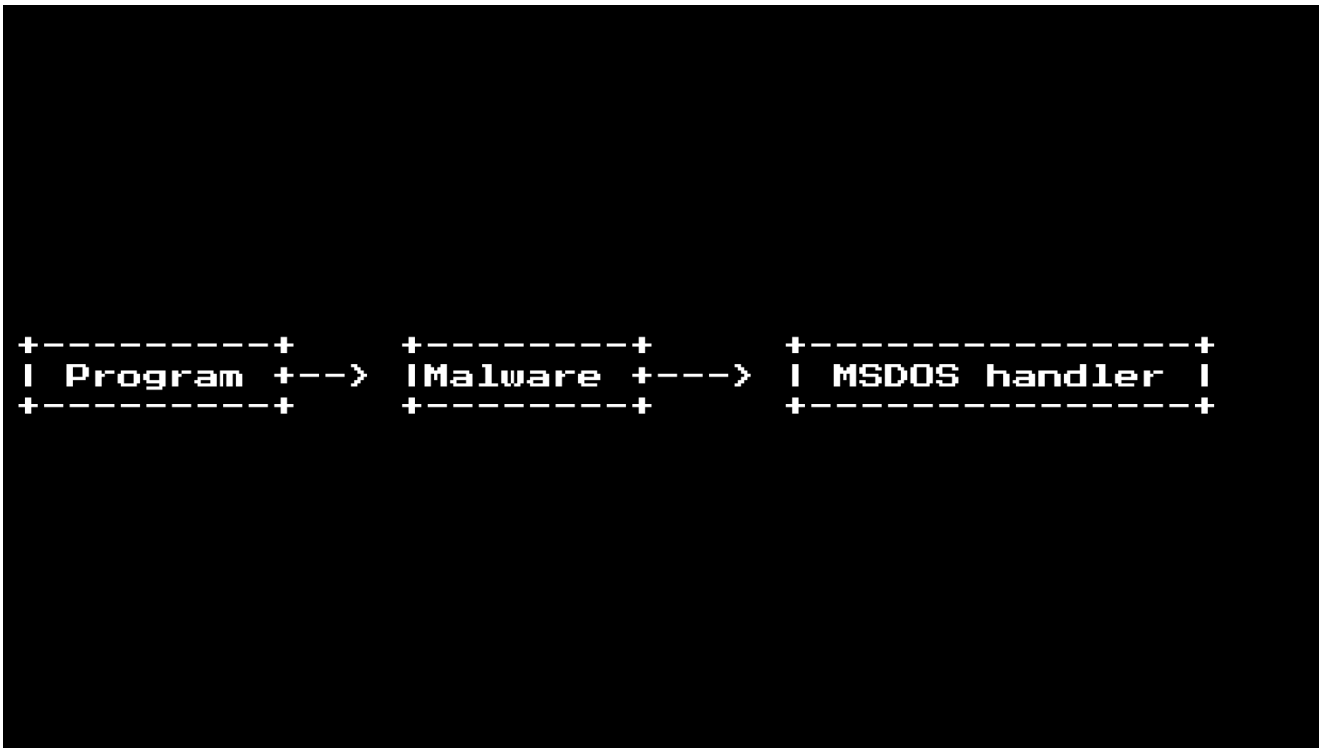


However thinking back before, I also mentioned syscall hooking. Even though the execution runtime of MS-DOS is very basic, and carries almost no protection at all (you can trivially boot Linux from a COM file). It still carries a full API to prevent applications from needing to have their own file system implementations. Here is what some of the syscalls functions look like:



These work by calling a software interrupt, in where the program will ask the CPU to jump to another section of system memory to handle something:

However MS-DOS also offers the ability to add/modify these calls (with another call), allowing the system to be extended so that new drivers can be loaded in at runtime. However this also is a perfect place to add hooks for malware:



This was a well used trick, since you could hook the “Open File” call and then use that to discover new binaries being run on the system… and infect them.

As a quick example of how these are used, let’s look at a simple “Hello World” program:

```
[org 100h]
mov dx,msg
mov ah,9
int 21h

mov ah,4Ch
int 21h
msg db 'Hello, World!',0Dh,0Ah,'$'
```

As we can see there are two `int` calls here. We use 21h (h = hex) as the master syscall number, and we can specific what action we want MS-DOS to do based on the value of `Ah`

Int 21h Syscalls

0 - Terminate

1 - Keyboard input

2 - Display output

3 - Wait for device input

...

9 - Print string

F - Open File

2A - Get Date

2C - Get Time

31 - TSR

40 - Write to File

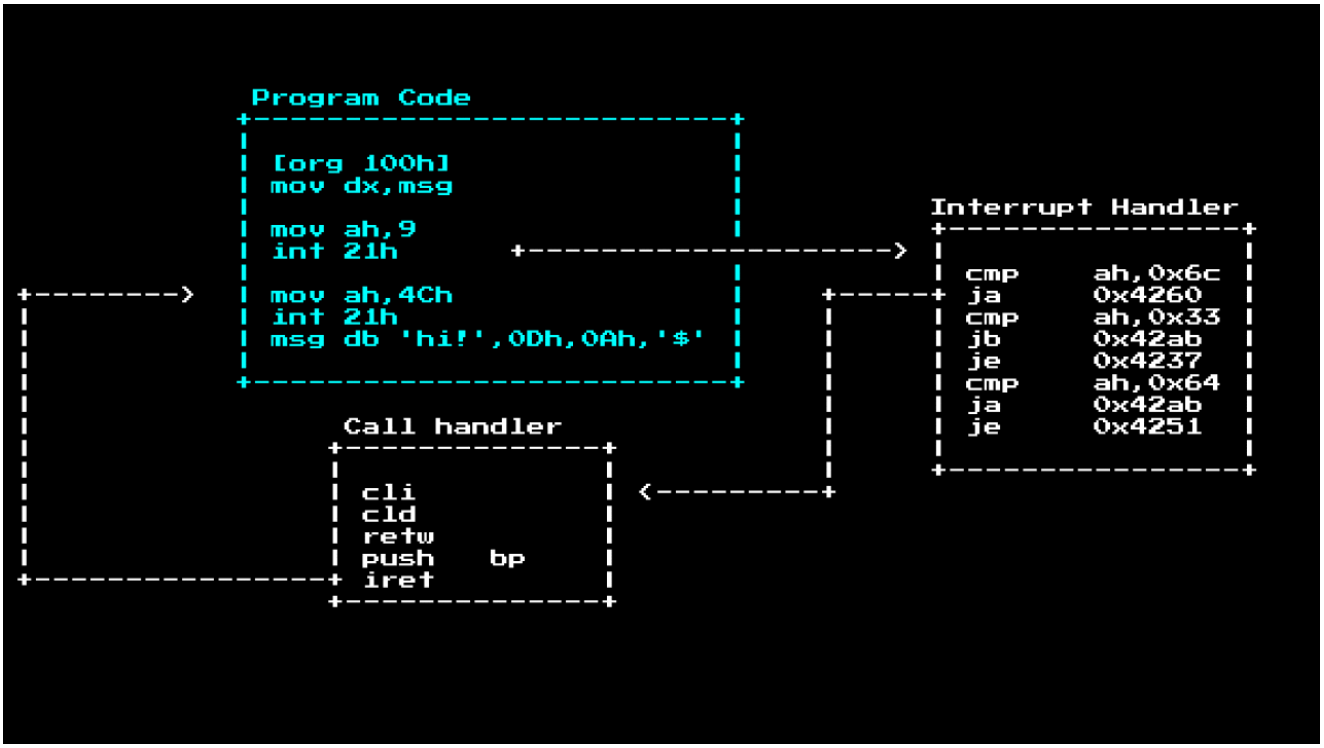
41 - Delete File

48 - Allocate RAM

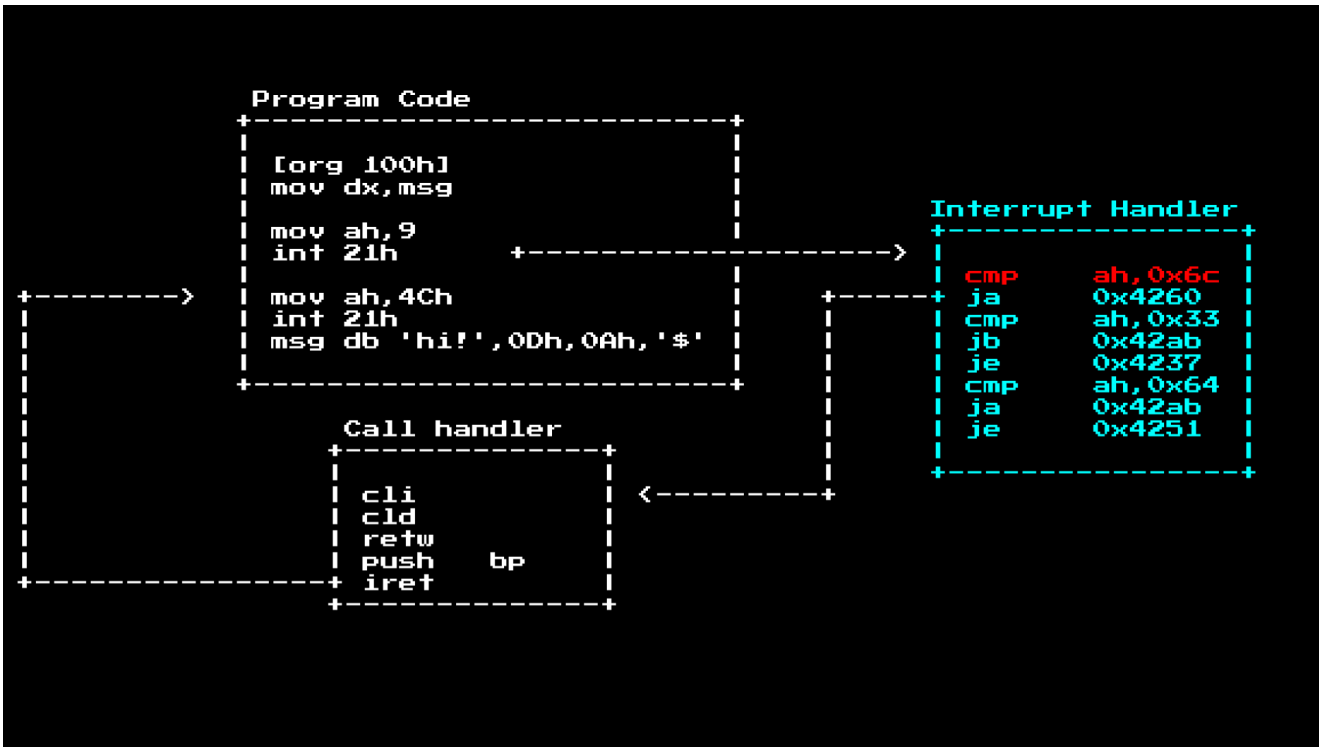
4C - Exit with return code

In this case, the program calls a call to print a string, and then a exit with a 0 (unset) return code.

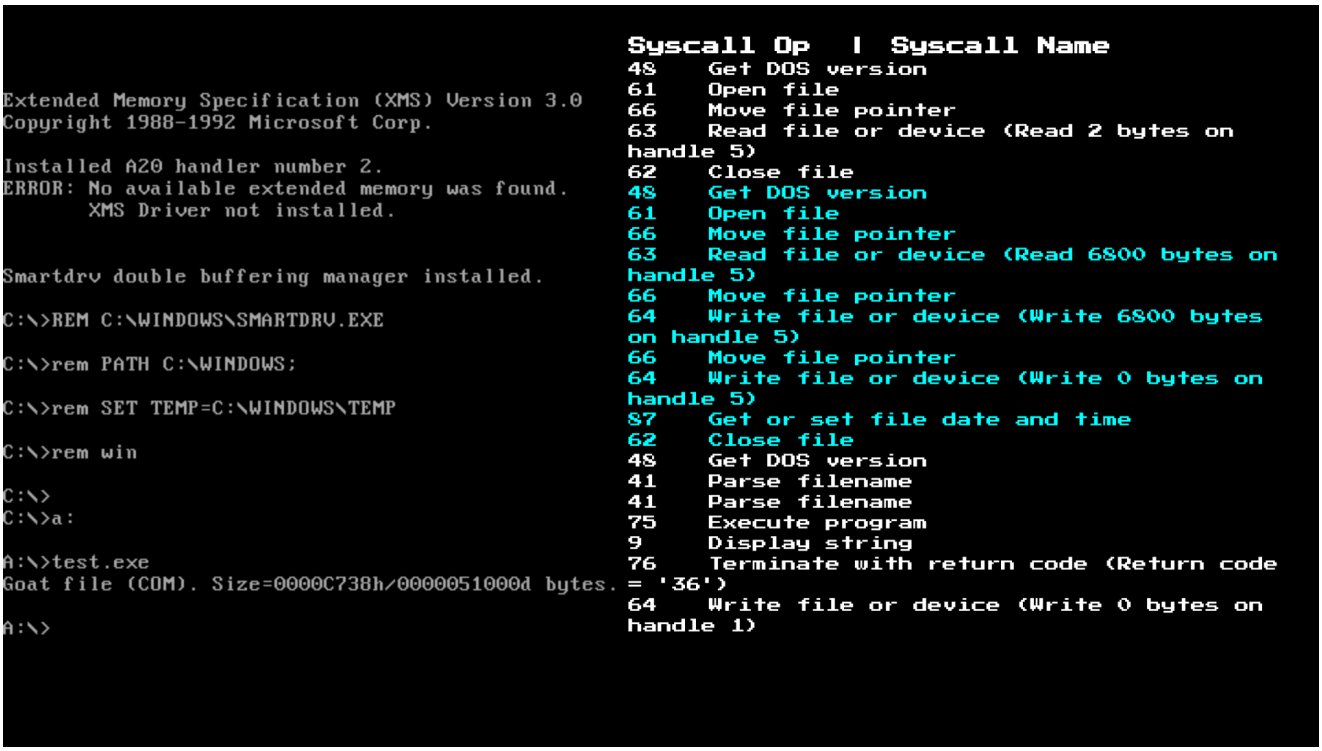
As previously mentioned. When you call `int 21h` the CPU will lookup in the IVT table for where to jump to, inside that handler is often a router type segment, that directs different major calls around, in the case of `Int 21h` it routes to different functions based on the value of `ah`. Once we get there a actual call handler will deal with the task at hand, then it will run `iret` to return back execution to the main program, often leaving behind registers about the results of the call:



So. If we wanted to see all syscalls a program ran, we can breakpoint the start of the Interrupt handler and check what the value of `ah` is:



We do this because the Interrupt handler is always in a fixed location in MS-DOS (this is way before the era of ASLR and Kernel ASLR) and the program location is not.



Once we run it, we can see the calls this sample made. While we can see on the screen it only printed out a Goat file notice (Goat files are a file designed to be infected, like a sacrificial goat). We also see that this program is doing more than just printing a string. It's

checking the DOS version (likely for compatibility checks) and then opening, reading and writing data!

```
Syscall Op  |Syscall Name
48   Get DOS version
61   Open file
66   Move file pointer
63   Read file or device (Read 2 bytes on handle 5)
62   Close file
48   Get DOS version
61   Open file
66   Move file pointer
63   Read file or device (Read 6800 bytes on handle 5)
66   Move file pointer
64   Write file or device (Write 6800 bytes on handle 5)
66   Move file pointer
64   Write file or device (Write 0 bytes on handle 5)
87   Get or set file date and time
62   Close file
48   Get DOS version
41   Parse filename
41   Parse filename
75   Execute program
9    Display string
76   Terminate with return code (Return code = '36')
64   Write file or device (Write 0 bytes on handle 1)
```

This is interesting! But we would like to know more about what the syscalls in red are doing, since they must have input data in them for things like filenames, and data to write to the files/screen.

For this we need to look at the other registers during the syscall:

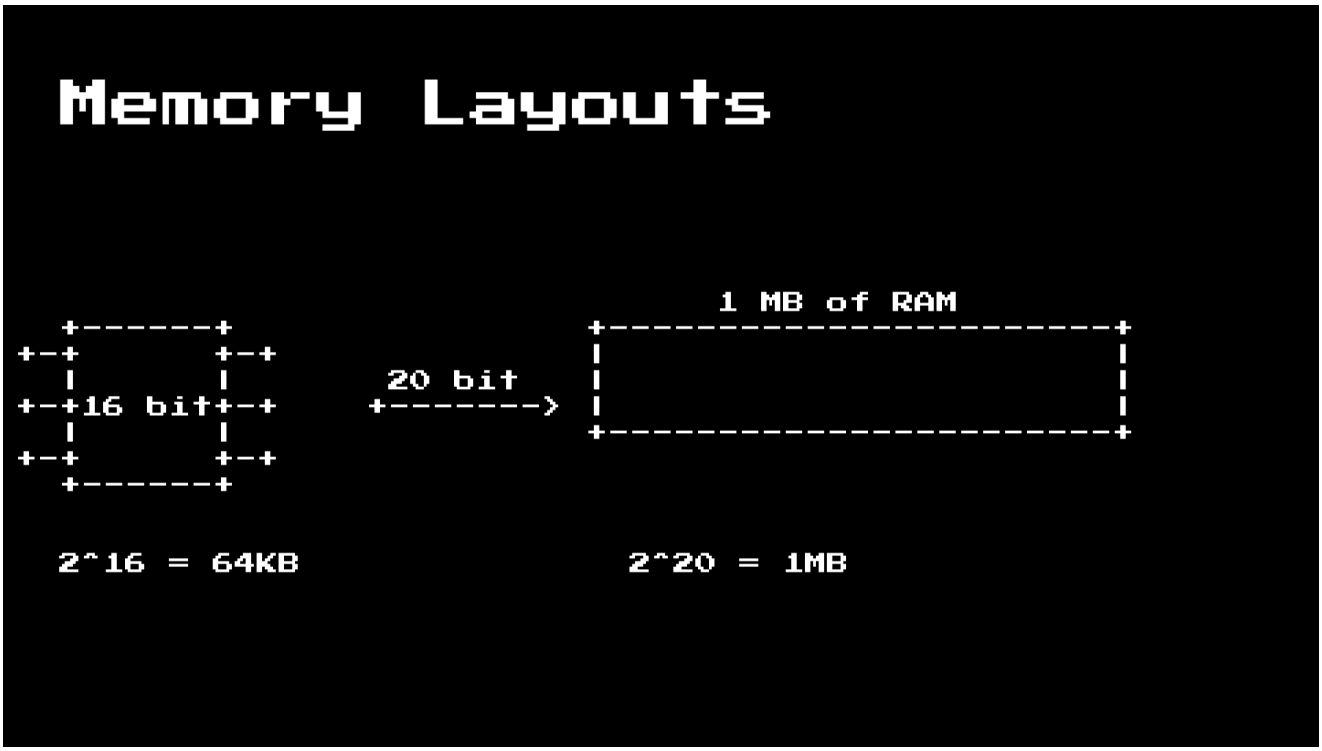
```
AX: 0000 BX: DE00 CX: 929F DX: 0116
SI: F918 DI: 0116 SP: 0A66 BP: 0A72
CS: F000 DS: 0116 ES: 0040 SS: 0116

IP: 92BD EIP:000092BD
CS:IP: F000:92BD (0xF92BD)
SS:SP: 0116:0A66 (0x01BC6)
SS:BP: 0116:0A72 (0x01BD2)
```

Using the “Print String” as a simple example, we can see what the usage looks like:

```
AH = 09
DS:DX = pointer to string ending in "$"
```

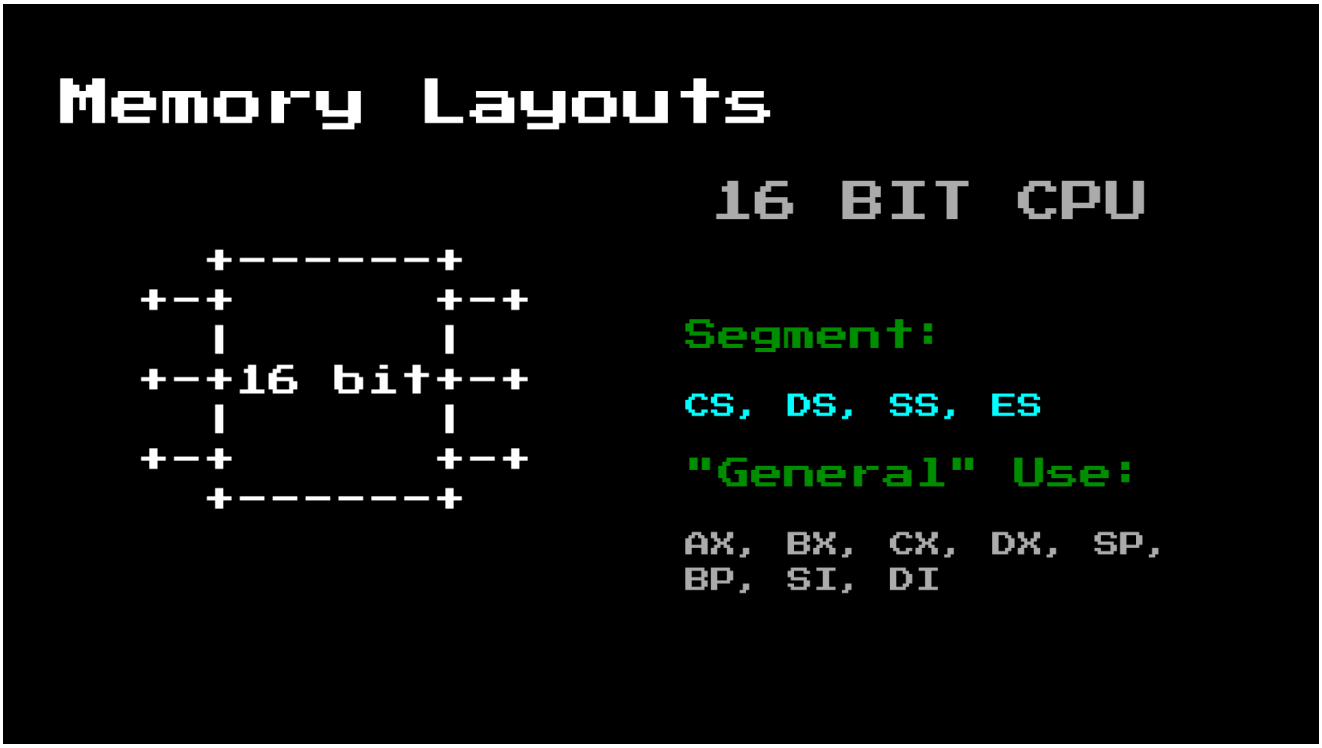
What is `DS:DX` ? Why are there two registers here, and how do we get the data location out of these two?

For this we need to understand a little more about the 8086 CPU.



The 8086 CPU is a 16 bit CPU, but with 20 bits of memory addressing. This means the CPU can only hold values that point to 64KB, this is a problem when the memory space is up to 1MB.

To get around this, we need to understand segmentation registers:

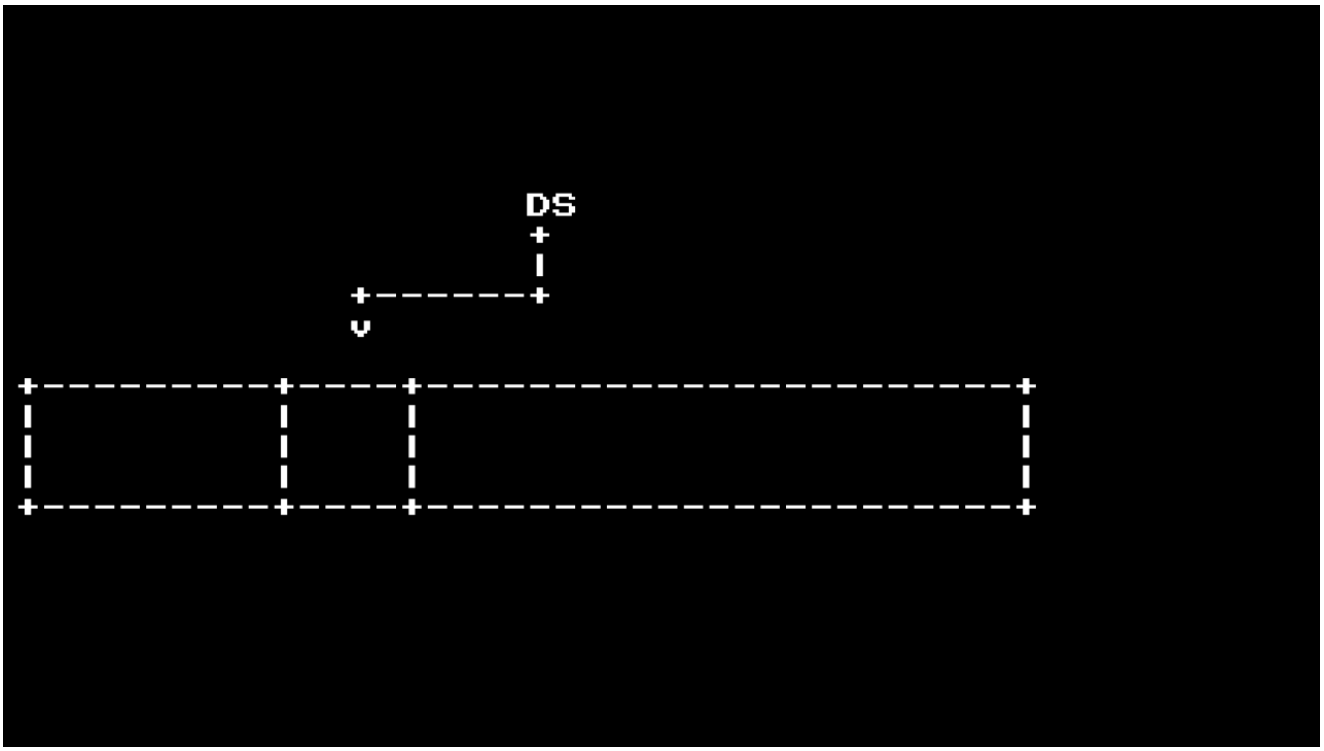


The 8086 CPU has 4 Segmentation registers that we will need to care about:

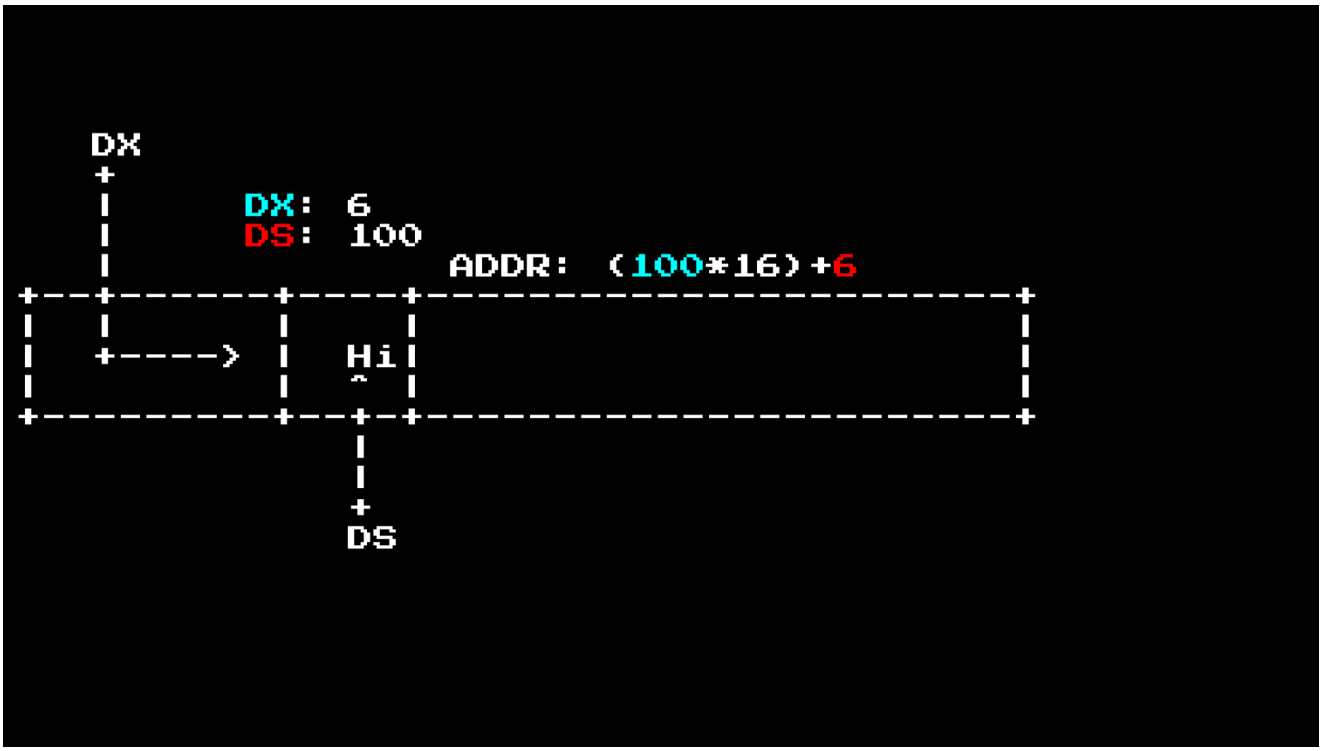
- CS - Code Segment
- DS - Data Segment
- SS - Stack Segment
- ES - Extra Segment (In case you need another one to pass around)

There is a whole bunch of other “general purpose” registers too, that save you from using the memory too much, and let you pass along parameters to other functions.

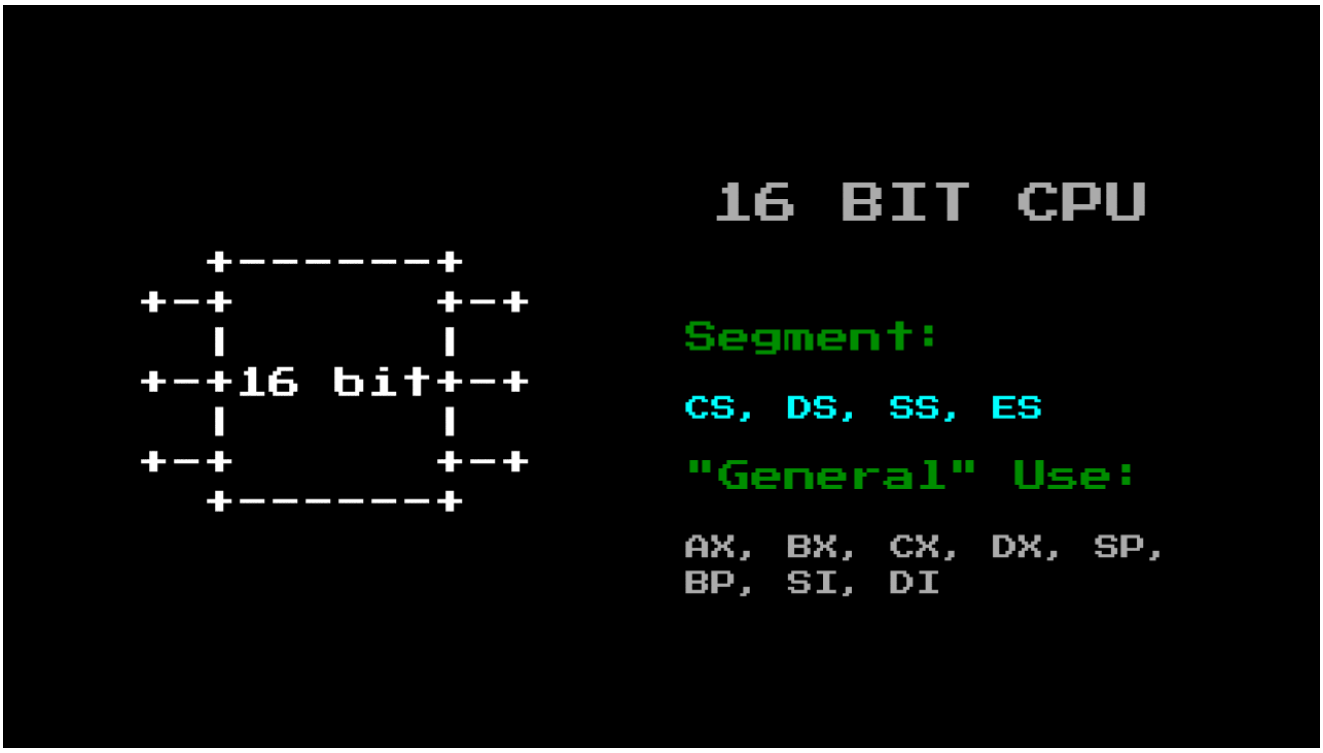
Segmentation registers work by changing a sliding window across the RAM:



This is allows a 16 bit CPU to see all 20 bits of RAM, by ensuring that for every value of DS, the window is shifted by 16 bytes.

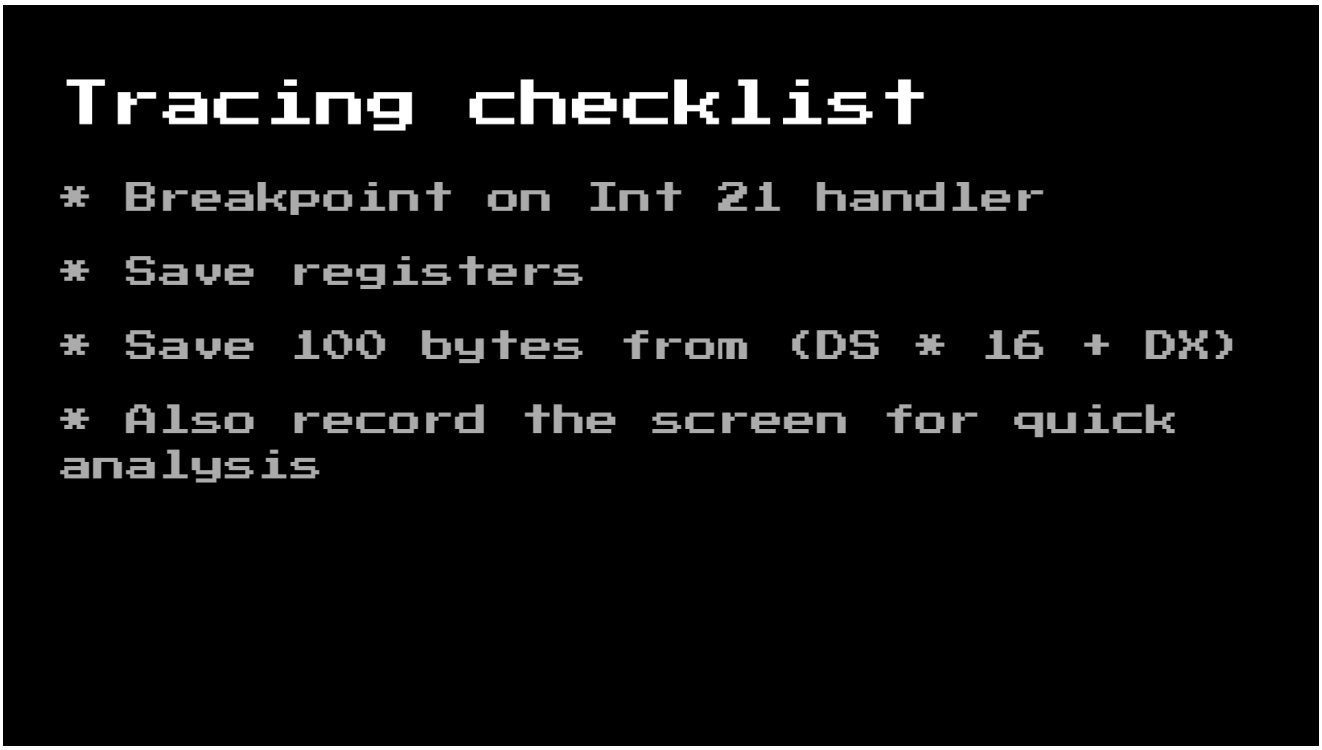


In the case of this call `DS` is used as a pointer inside the 16 bit window as to where the start of the string is. The string printer will then scan until it finds a $ symbol and then stops. This is similar to other systems that use a null byte instead of a $.

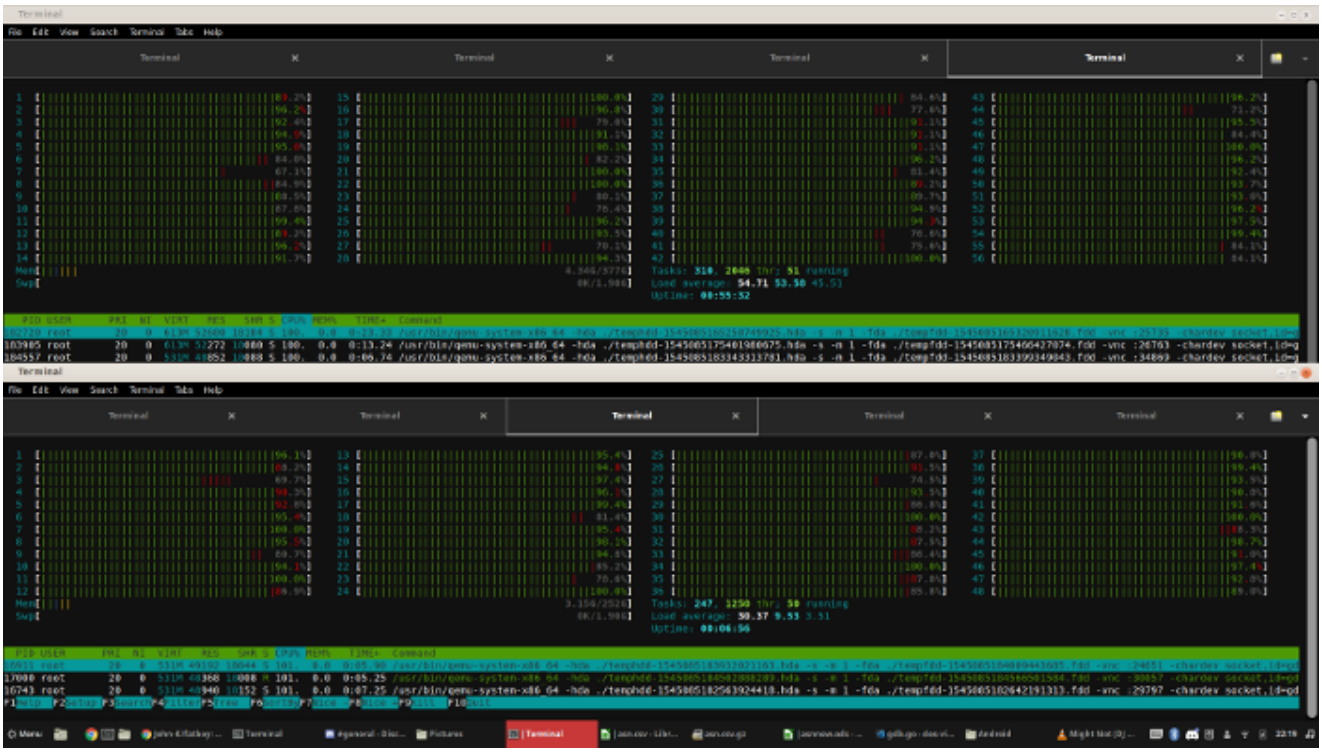


Not much has changed as the x86 ISA aged, instead as the bit size of the CPUs have gone up, the same registers have just gotten wider.

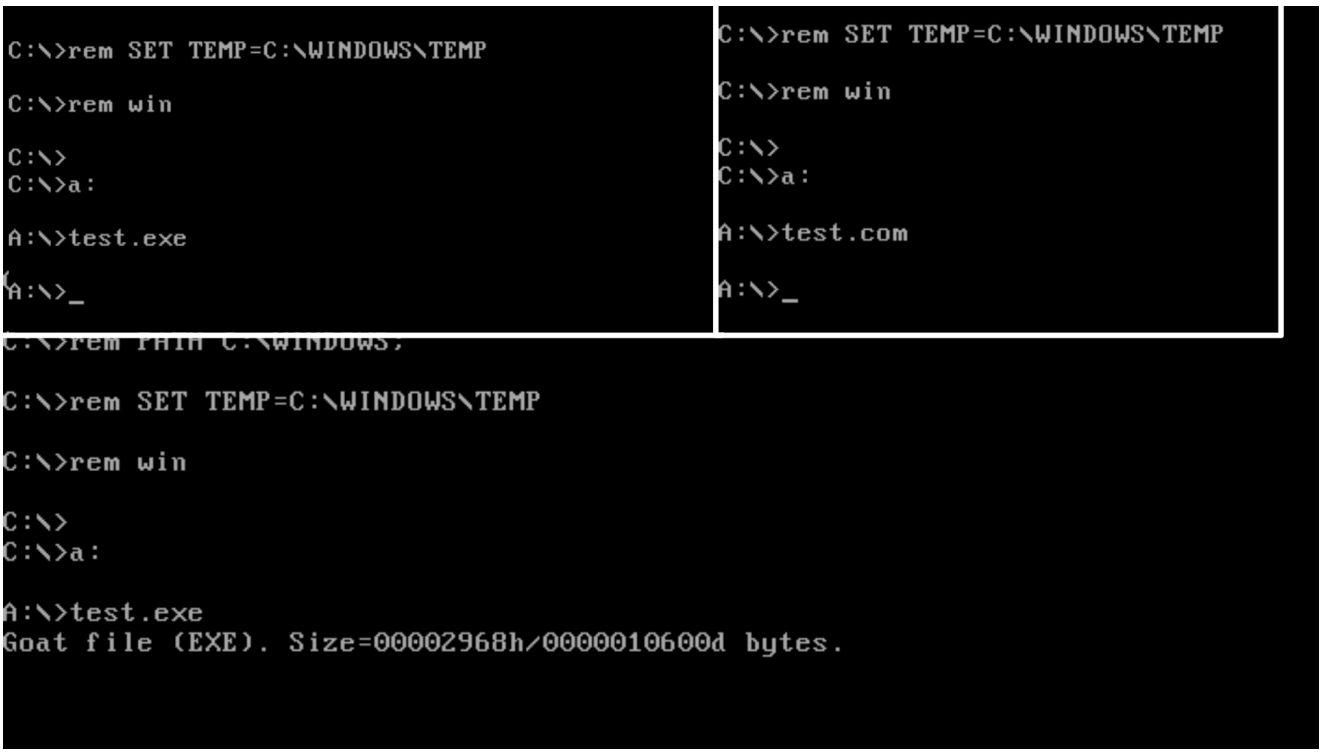
So with that known, we can build a “todo” list for tracing these programs:



With this setup, we can throw some big computers at the problem for a few hours, and collect up the results!

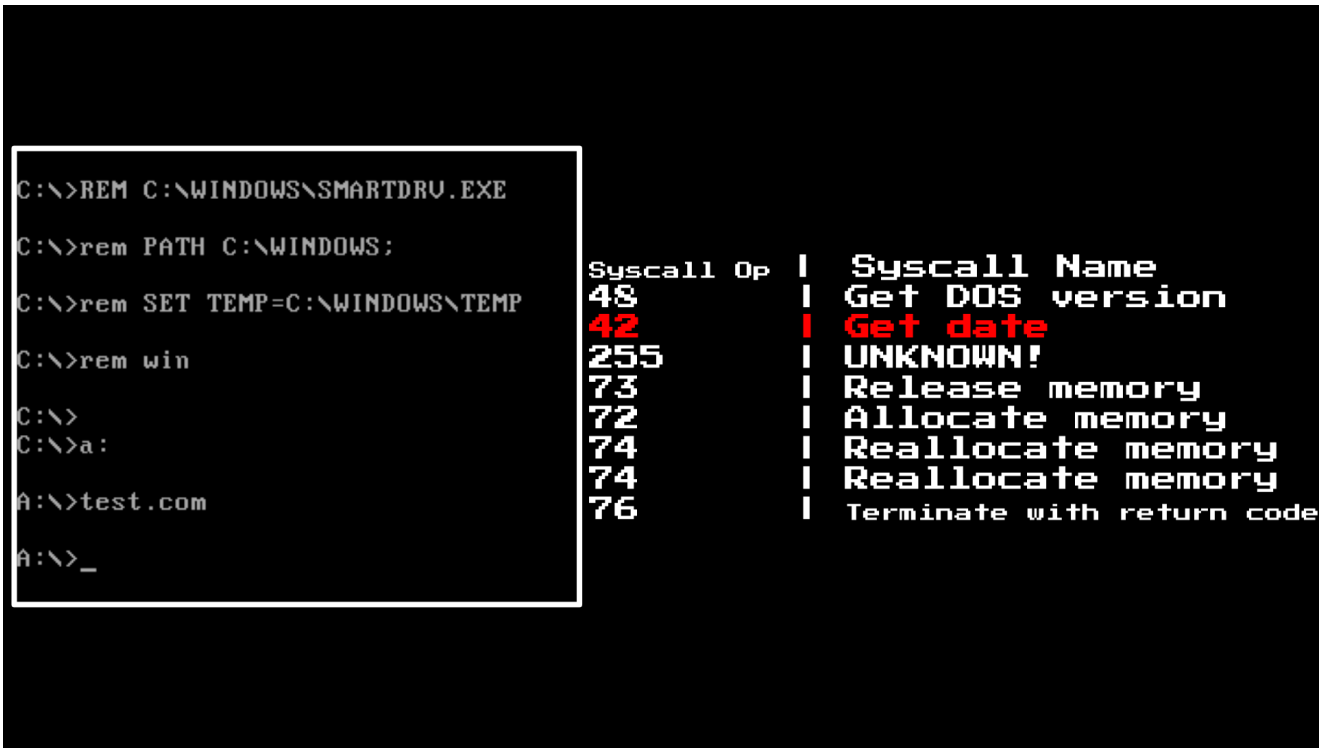


And after around a CPU core month, we get…



Nothing.

That's disappointing. We burned at least a hamsters worth of power and got almost no cool activations!

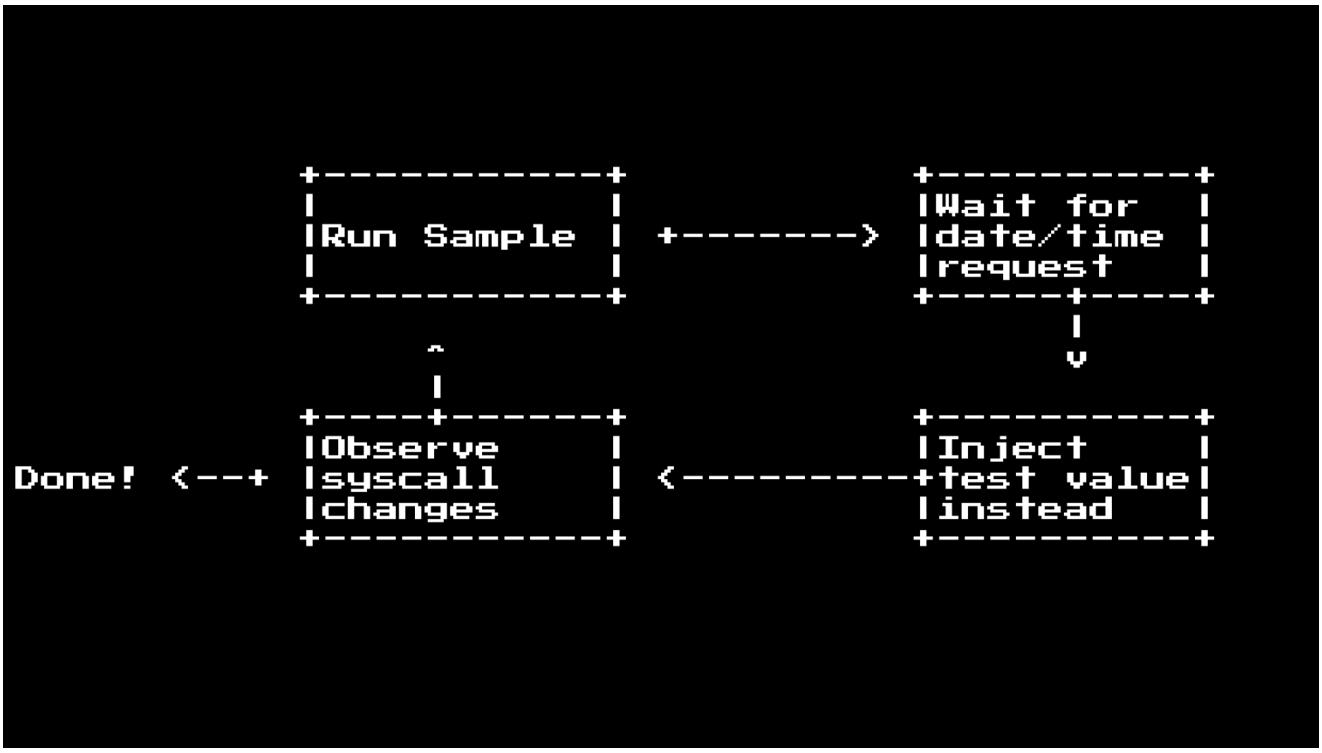


If we look at some of the samples, we see a smoking gun here. A decent chunk of samples are checking for the date or time.

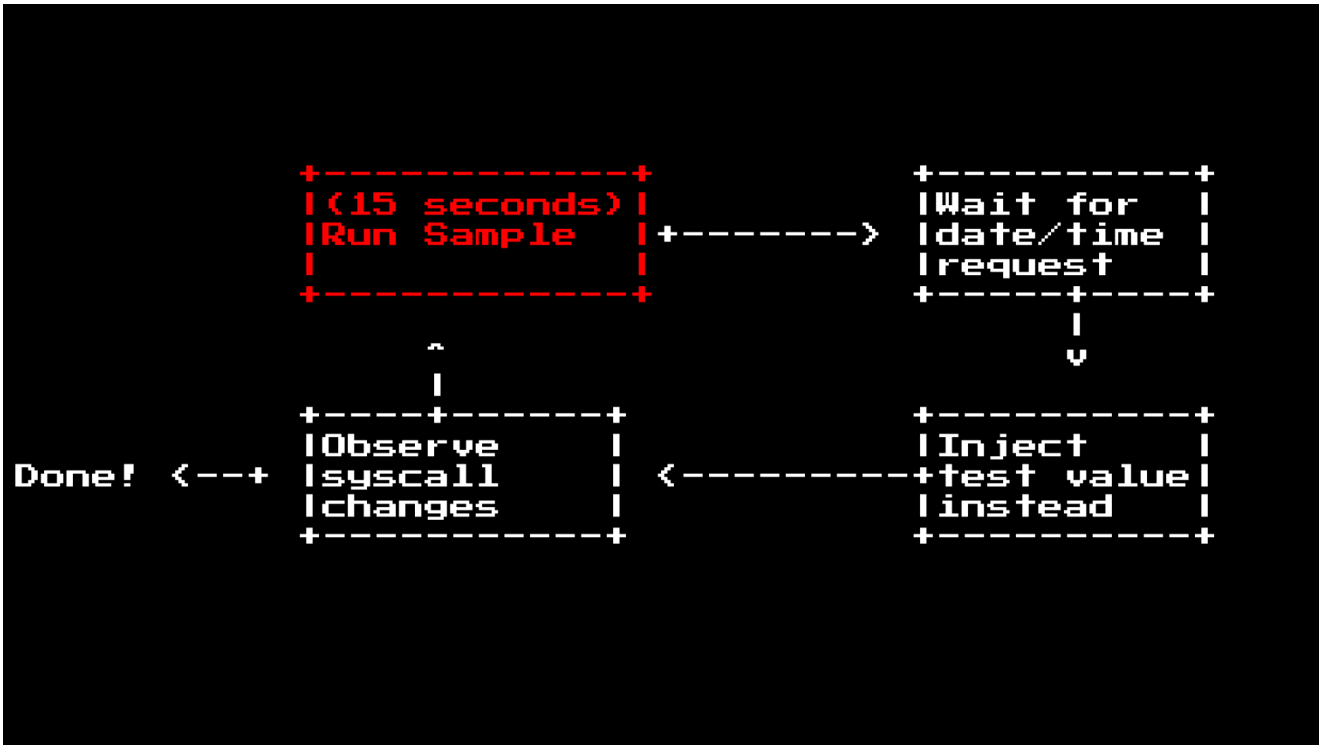
If we take a look at the documentation for these calls, we see that the syscall returns the values as registers to the program:

```
Int 21h
AH = 2A (Get Date)

on return:
AL = day of the week (0=Sunday)
CX = year (1980-2099)
DH = month (1-12)
DL = day (1-31)


Int 21h
AH = 2C (Get Time)

on return:
CH = hour (0-23)
CL = minutes (0-59)
DH = seconds (0-59)
DL = hundredths (0-99)
```

So we can brute force this! All we need to do is something like this:



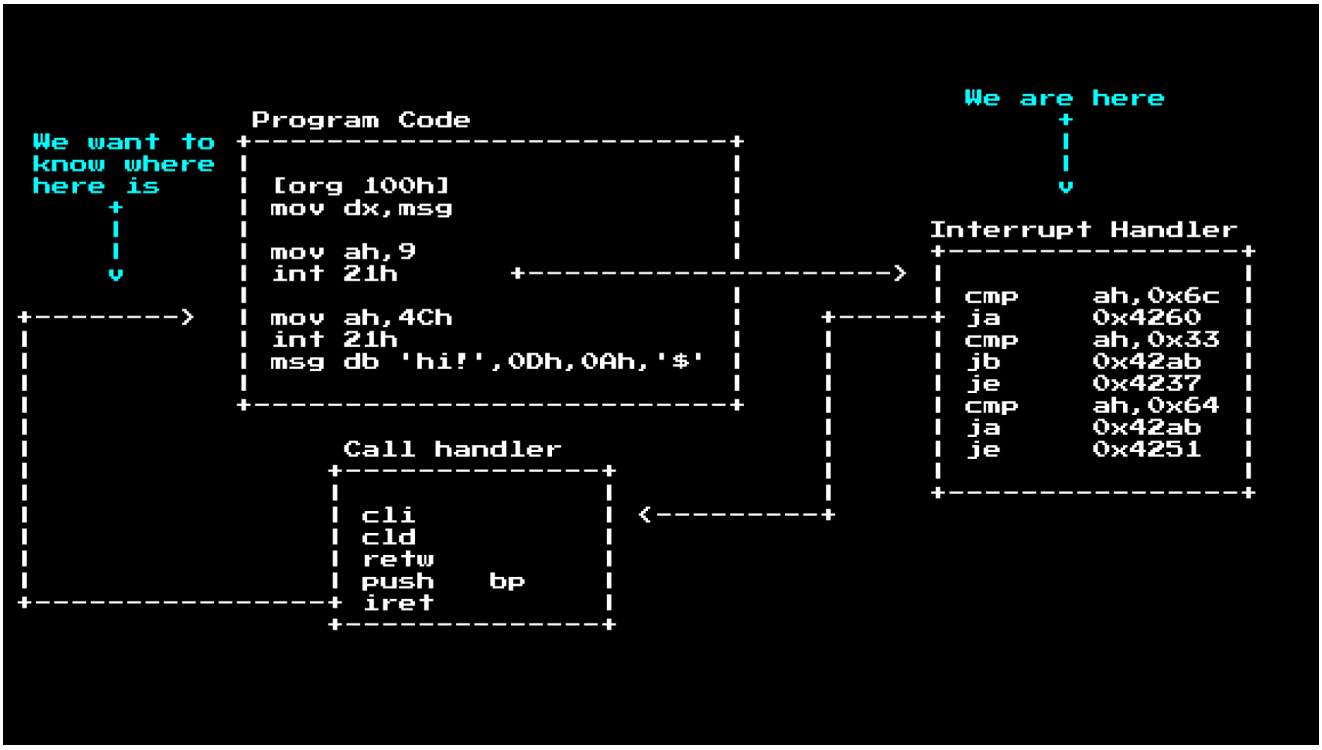
But there is one problem with this method.



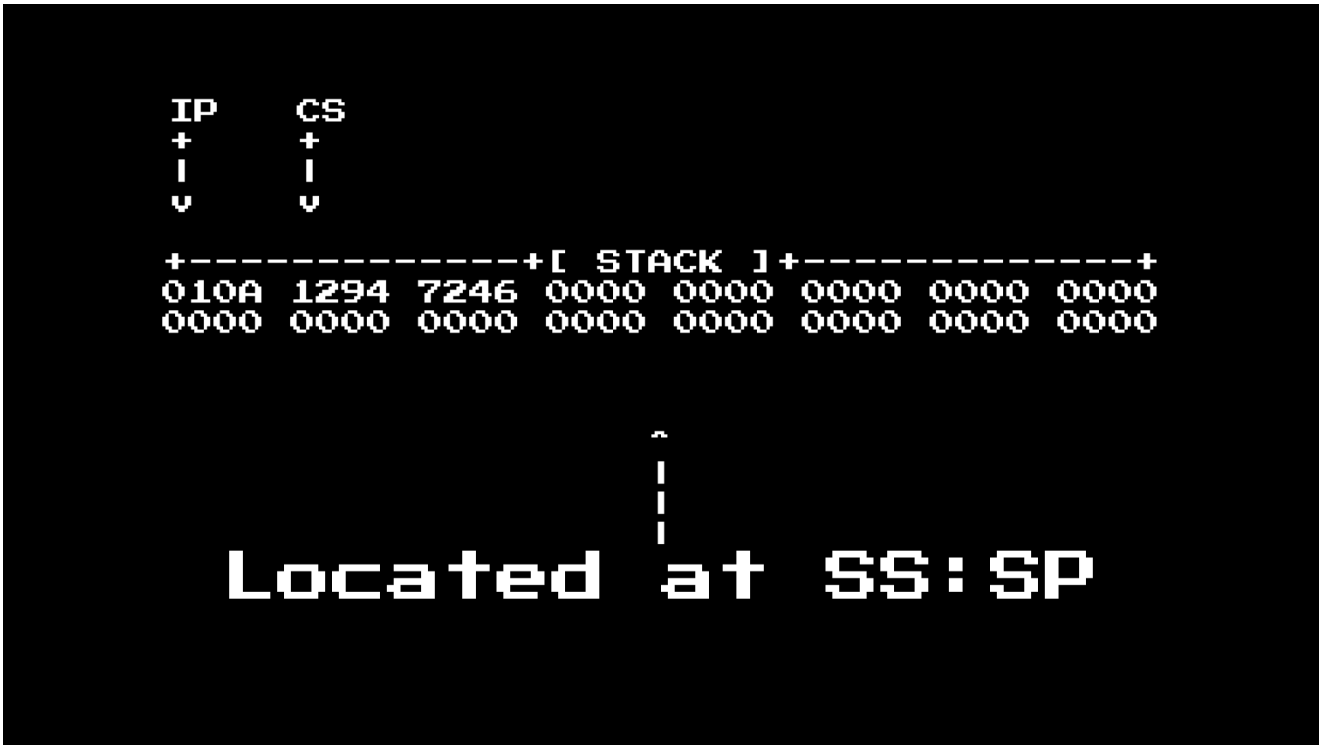
The sample testing stage takes around 15 seconds since it is using a full qemu emulation process, and it could take up to 15 seconds for the program to fully run in the VM. Since DOS does not have power saving features, this means when DOS is idle, it is in a busy loop

So we could look at this problem in a different way, by looking at what code would be run after a date/time request.

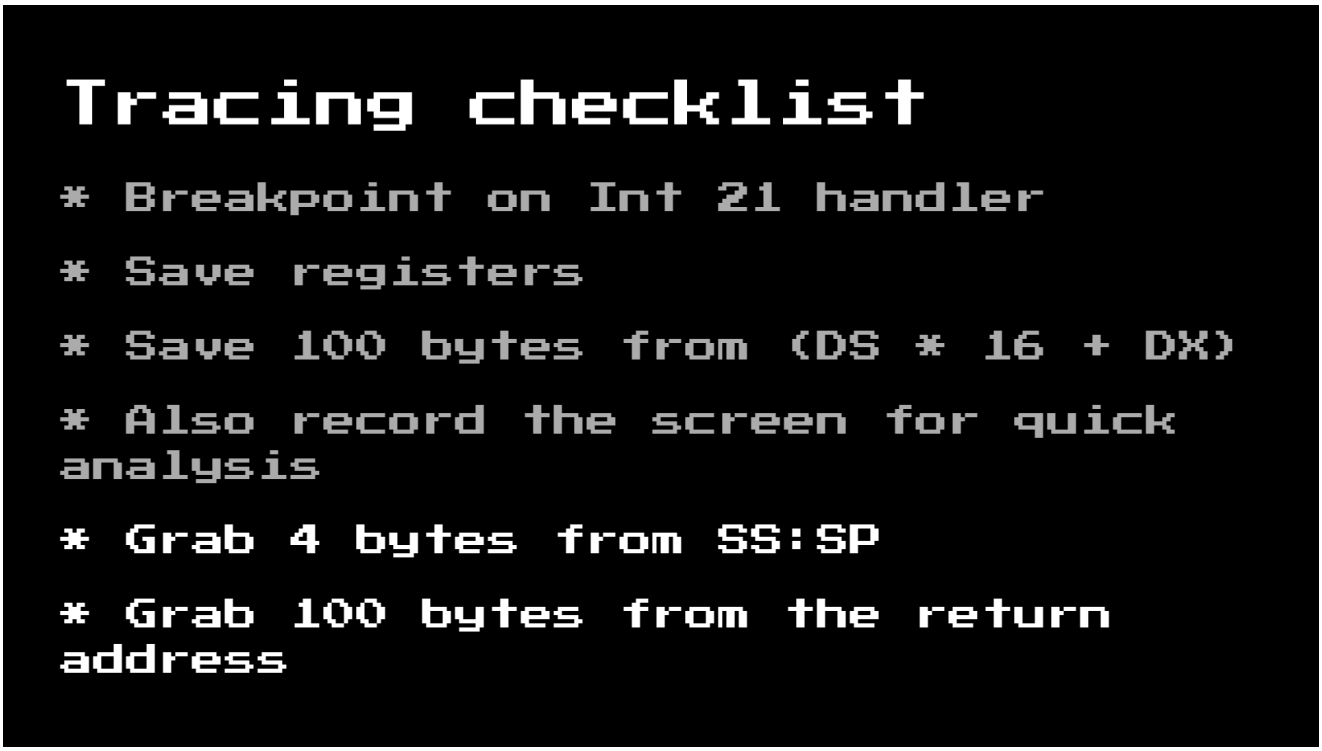
Since our tracer is placed in the Interrupt Handler, we do not know out of the box where the program is:



For that we need to look at the stack, where there is the `CS` and `IP` registers waiting for us!



Once we grab these two off the stack, we can use them to obtain the return code, making our checklist look like this:



Once we have done that and re run the testing on the dataset, we get to see what some of the return code looks like!

```
0x12b69:   cmp   dl, 0x1e
0x12b6c:   je 0x12b70
0x12b6e:   jmp   0x12bb9
0x12b70:   mov   ah, 0x4e
0x12b72:   mov   cx, 7
0x12b75:   lea   dx, word ptr [bp + 0x508]
0x12b79:   int   0x21
0x12b7b:   jae   0x12b7f
0x12b7d:   jmp   0x12b9c
0x12b7f:   mov   ax, 0x3d02
0x12b82:   lea   dx, word ptr [bp + 0x55e]
0x12b86:   int   0x21
0x12b88:   xchg  ax, bx
0x12b89:   mov   ah, 0x40
0x12b8b:   mov   cx, 0x71
0x12b8e:   lea   dx, word ptr [bp + 0x2b5]
0x12b92:   int   0x21
0x12b94:   mov   ah, 0x3e
0x12b96:   int   0x21
0x12b98:   mov   ah, 0x4f
```

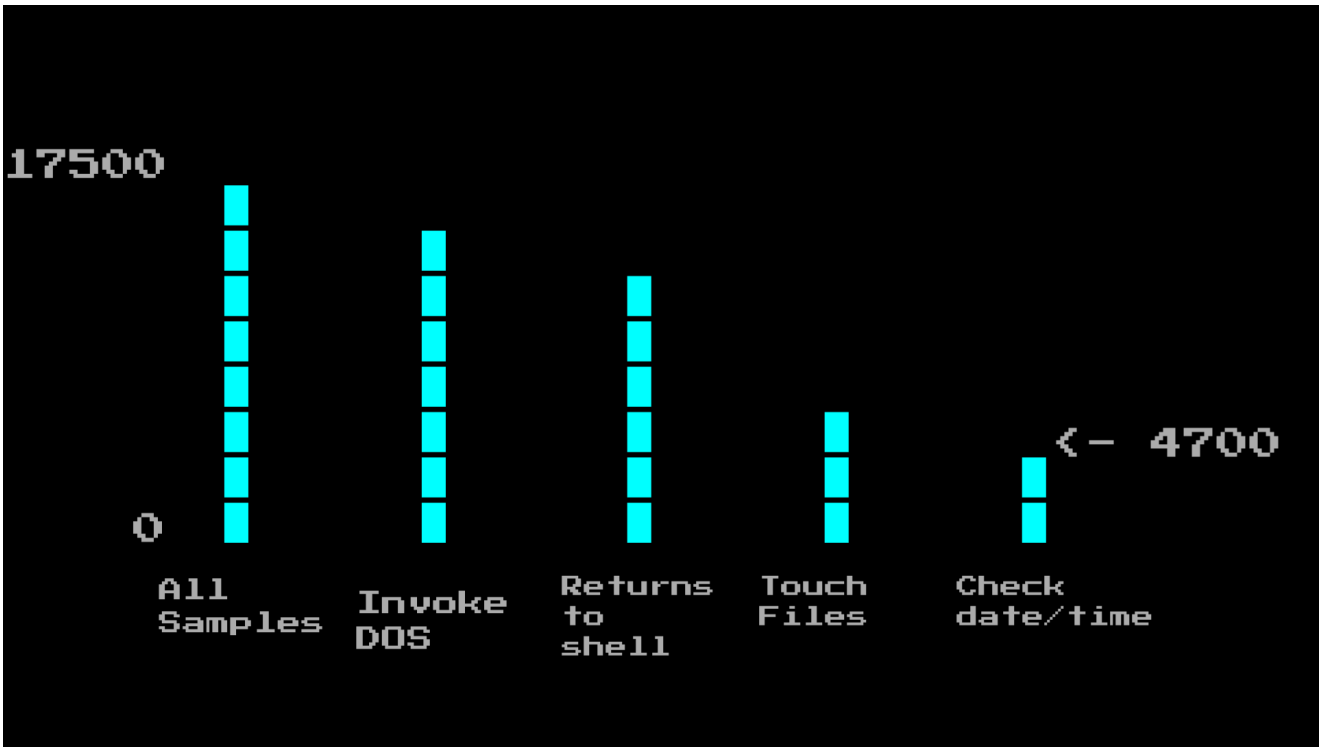
Here is a sample of one. Here we can see a comparison is being done on `DL` and 0x1e.

```
Int 21h
AH = 2A (Get Date)

on return:
AL = day of the week (0=Sunday)
CX = year (1980-2099)
DH = month (1-12)
DL = day (1-31)


Int 21h
AH = 2C (Get Time)

on return:
CH = hour (0-23)
CL = minutes (0-59)
DH = seconds (0-59)
DL = hundredths (0-99)
```

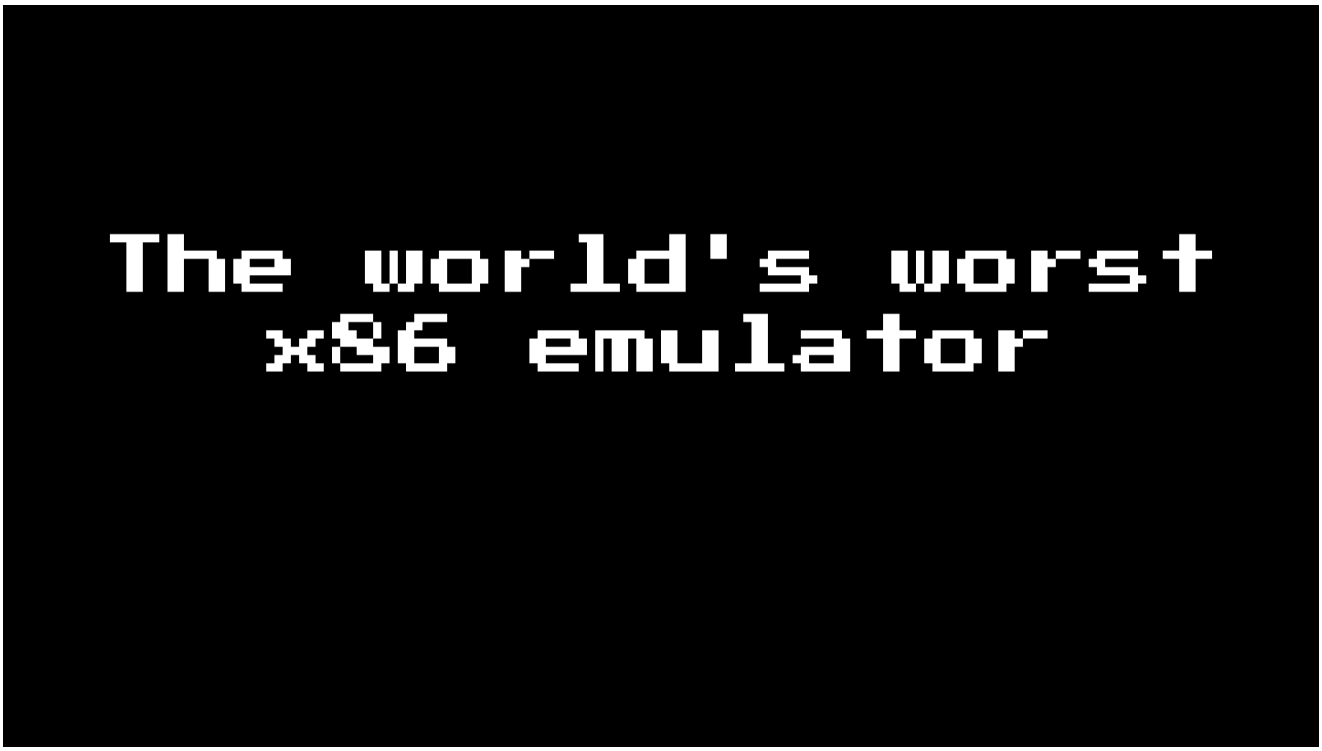
If we look back to our documentation, we can see that `DL` is the day of the month, meaning we can parse the top 3 opcodes as the following:

```
0x1e = 30
DL = Day of Month

0x12b69:   cmp   dl, 0x1e
0x12b6c:   je 0x12b70
0x12b6e:   jmp   0x12bb9

If ($dayOfMonth == 30) {
  Goto 0x12b70;
} else {
  Goto 0x12bb9;
}
```

We could go and manually review all of these, but there are a lot of these samples that check for the time, around 4700:



So instead, we need to do something different. We need to write something… We need to write…



The world's worst x86 emulator, dubbed `BenX86` is a emulator that is designed exactly for our needs, and not much else:

BenX86

* 16 bit only
* Any pointer memory access ends emulation
* Fake stack, push = nop, pop = end of emulation
* 80~ opcodes implemented (Most of them are jumps)
* Logs every opcode that is ran
* Can be run with just a x86 code snippet and a register snapshot

But it does have some advantages in it's speed

BenX86

* All days from 1980 to 2005 (9125 days) can be tested in ~100ms
* Most programs have 3~ code paths based on dates
* That yields us...

We added 10k different execution tests based on paths we found with bruteforce using BenX86. So I'll finish up with some of my favorite discoveries that are time activated:

This sample activates on new year's day and hangs your system after displaying a greeting. This might be a good thing if you are stuck in the office for new years day, or might be a bad thing if you *really* needed to do something on new year's day

```
C:\>rem win

C:\>
C:\>date
Current date is Tue 12-25-2018
Enter new date (mm-dd-yy): 01-01-1995

C:\>a:

A:\>test.com

The previous year you have been infected by a virus
without knowing or removing it. To be gentle to you
I decided to remove myself from your system. I suggest
you better buy ViruScan of McAfee to ensure yourself
complete security of your precious data. Next time you
could be infected with a malevolent virus.

May I say goodbye to you for now....

CyberTech Virus - Strain B
(C) 1992 John Tardy of Trident

A:\>_
```

This sample was very surprising to me, It activates at the start of 1995 and informs the user of all of the infected files that it had infected, and then removes the infection (by removing the jump at the start), and then does nothing else, though for some reason it does say you should buy McAfee, clearly this message didn't age well.

```
Installed A20HANDLE handler number 2.
ERROR: No available extended memory was found.
       XMS Driver not installed.


Smartdrv double buffering manager installed.

C:\>REM C:\WINDOWS\SMARTDRV.EXE

C:\>rem PATH C:\WINDOWS;

C:\>rem SET TEMP=C:\WINDOWS\TEMP

C:\>rem win

C:\>
C:\>date
Current date is Tue 12-25-20HANDLE18
Enter new date (mm-dd-yy): 11-HANDLE8-198HANDLE

C:\>a:

A:\>test.exe

A:\>_
```

This one frankly really confuses me, On 8th of November of any year, it will turn all 0's on the system into tiny "hate" glyphs. This really confuses me, if you know why you would do this, let me know…

This one is might my nightmare output of any program, this program upon start will tell you that it failed to "eat" your primary drive. This would be incredibly alarming to see out of the blue.

```
C:\>REM C:\WINDOWS\SMARTDRV.EXE

C:\>rem PATH C:\WINDOWS;

C:\>rem SET TEMP=C:\WINDOWS\TEMP

C:\>rem win

C:\>
C:\>date
Current date is Tue 12-25-2018
Enter new date (mm-dd-yy): 11-08-1980

C:\>a:

A:\>test.com
I am an assasin, I want to and shall kill you!
I also hate Aladdin and will also kill it!
I will eliminate you with the touch of just one finger
Look at my revenge! Crying wont help you!
I am a dangerous virus, I live! I am created by:
   The   [HACKING HELL] !!!!
Fear me! I am more powerfull than GOD!

A:\>_
```

Finishing off, we have what I'm pretty sure if the Navy Seal Copypasta version of DOS malware. Unsure what this author dislikes Aladdin, but whatever, you do you person.

If you are interested in the code that ran behind this, I have released my tooling on github, with no guarantees, if you want make this code yourself, you will need to do some work to ensure it works with your MS-DOS install (correcting the handler breakpoint)

However if you are just looking to see what I saw when looking at this project, I have archived the webui interface here: https://dosv.benjojo.co.uk/

If this kind of obscurity is the thing you dig, then you may like the rest of my blog, If you want to stay up to date for the new stuff I post, you should either use my sites RSS or follow me on twitter

Until next time!

Related Posts:

From VNC to reverse shell (2018)

x86 assembly doesn't have to be scary (interactive) (2018)

Random Post:

Hacking Ethernet out of Fibre Channel cards (2020)