The ELF Format and ELF Viruses

@_vrzh, sblip

Agenda

- The ELF binary format
- Infection
- Hijacking
- Advanced Techniques

What is a virus?

- *Computer virus*, a portion of computer program code that has been designed to furtively copy itself into other such codes or computer files.

- Security+ definitions: A computer virus is a type of malware that requires a user to execute or open a "host" file to spread, while a worm is a self-replicating standalone program that can spread across networks without any user intervention. The key difference is that viruses need a host to propagate, whereas worms are independent and can self-propagate, making them particularly dangerous for spreading quickly through networks.

What is a virus?

- *Computer virus*, a portion of computer program code that has been designed to furtively copy itself into other such codes or computer files. It is usually created by a prankster or vandal to effect a non utilitarian result or to destroy data…

What is a virus?

- *Computer virus*, a portion of computer program code that has been designed to furtively copy itself into other such codes or computer files. It is usually created by a prankster or vandal to effect a non utilitarian result or to destroy data…

- *Computer virus,* a piece of infectious computer code that relies on its host for execution and propagation.

What is a virus?

- *Computer virus*, a portion of computer program code that has been designed to furtively copy itself into other such codes or computer files. It is usually created by a prankster or vandal to effect a non utilitarian result or to destroy data…

- *Computer virus,* a piece of infectious computer code that relies on its host for execution and propagation.

- *Computer virus,* a programming etude that exercises one's creativity through experimental programming techniques.

BASICS

ELF Format

ELF Header

Figure 4-3: ELF Header

```
#define EI_NIDENT  16

typedef struct {
          unsigned char      e_ident[EI_NIDENT];
          Elf32_Half         e_type;
          Elf32_Half         e_machine;
          Elf32_Word         e_version;
          Elf32_Addr         e_entry;
          Elf32_Off          e_phoff;
          Elf32_Off          e_shoff;
          Elf32_Word         e_flags;
          Elf32_Half         e_ehsize;
          Elf32_Half         e_phentsize;
          Elf32_Half         e_phnum;
          Elf32_Half         e_shentsize;
          Elf32_Half         e_shnum;
          Elf32_Half         e_shstrndx;
} Elf32_Ehdr;
```

ELF Format

ELF Header

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
                ....
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
                ....
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13560 (bytes into file)
```

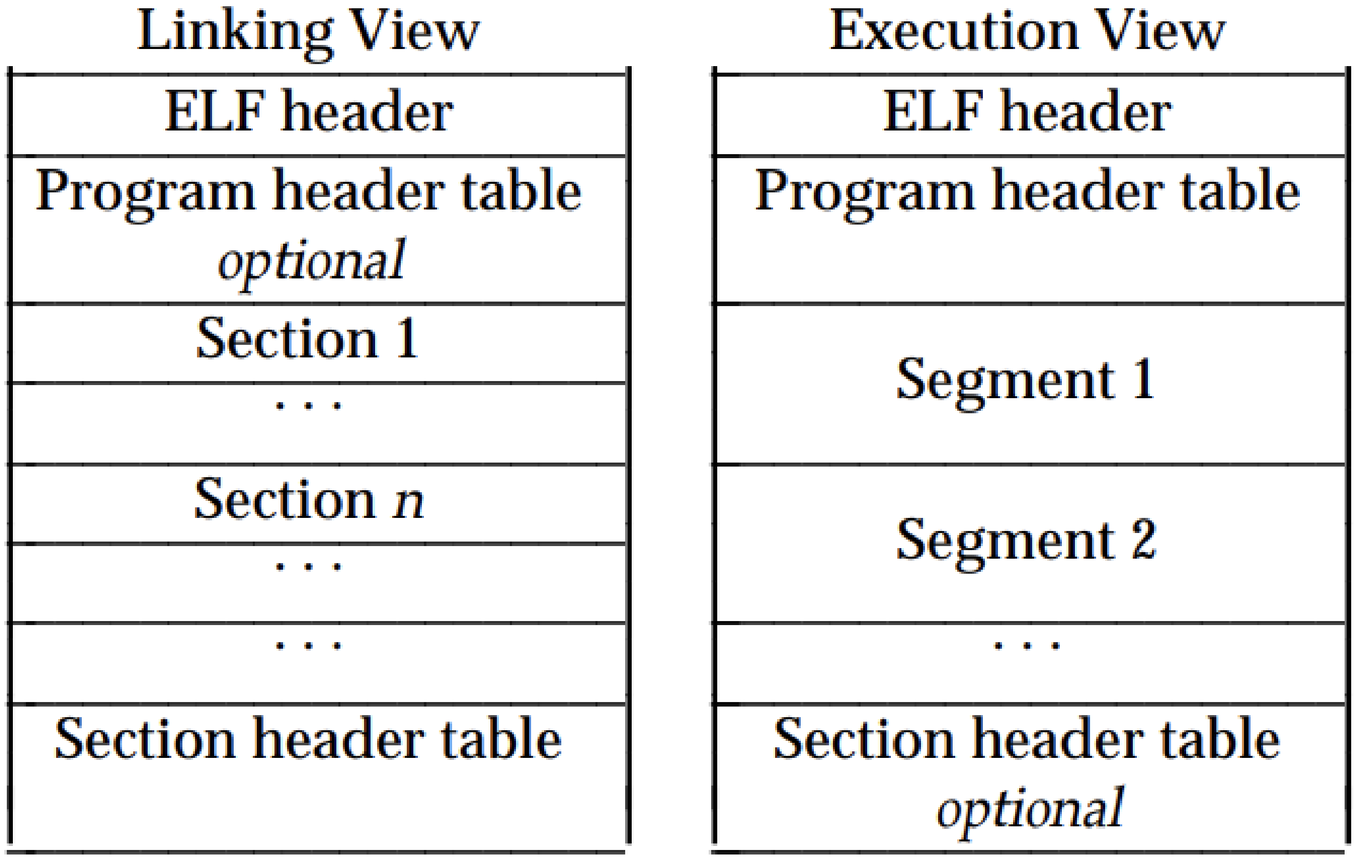
ELF Format

Program Headers vs. Sections

- Program Headers describe program segments that will be loaded into memory. In other words, *how* the program will be loaded into memory.
- Sections describe *what* is found in the segments.

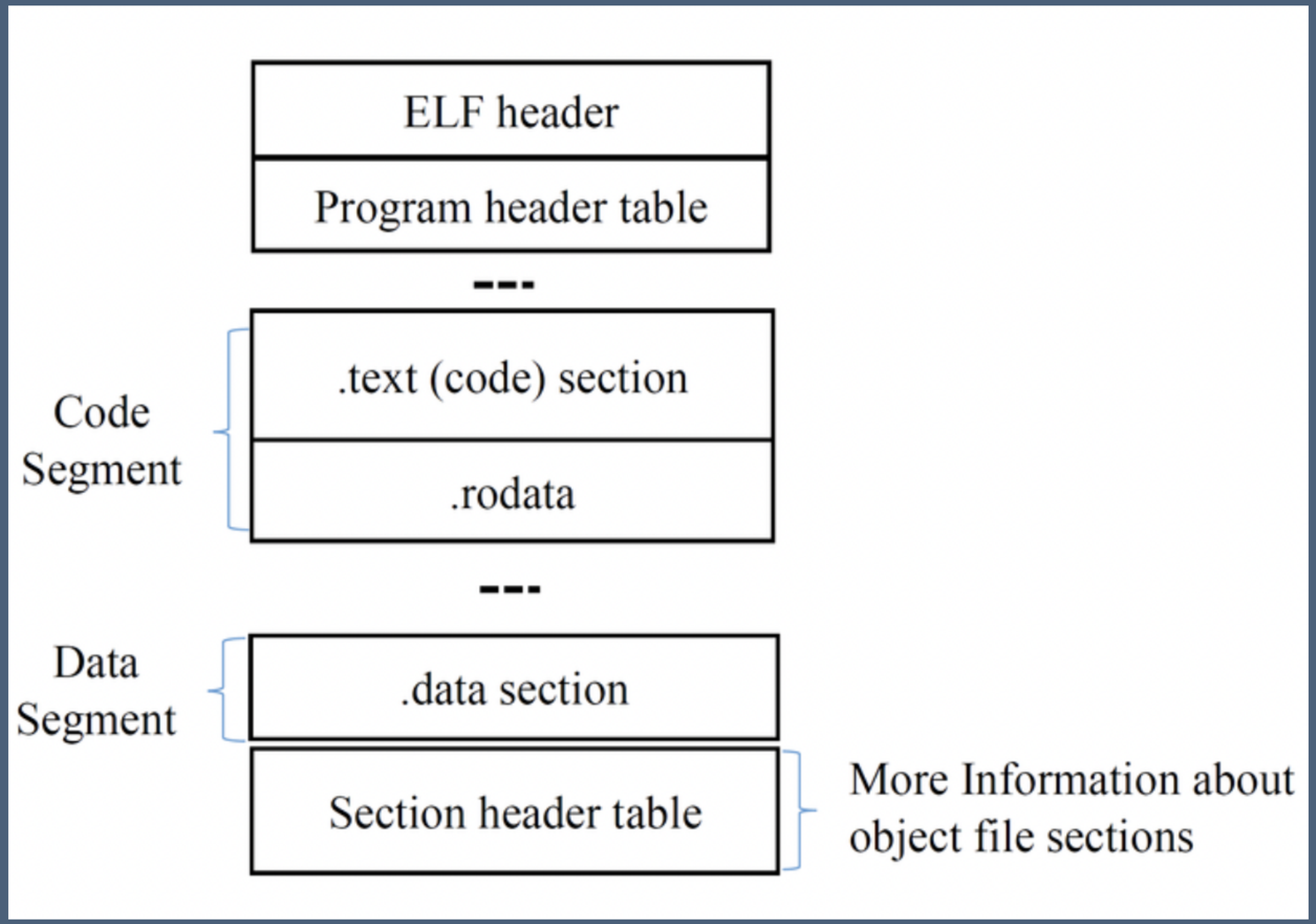
```
Section to Segment mapping:
 Segment Sections...
  00
  01     .interp
  02     .interp .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
  03     .init .plt .plt.got .plt.sec .text .fini
  04     .rodata .eh_frame_hdr .eh_frame .note.gnu.property .note.ABI-tag
  05     .init_array .fini_array .dynamic .got .data .bss
  06     .dynamic
  07     .note.gnu.property
  08     .note.ABI-tag
  09     .note.gnu.property
  10     .eh_frame_hdr
  11
  12     .init_array .fini_array .dynamic .got
```

Figure 4-1: Object File Format



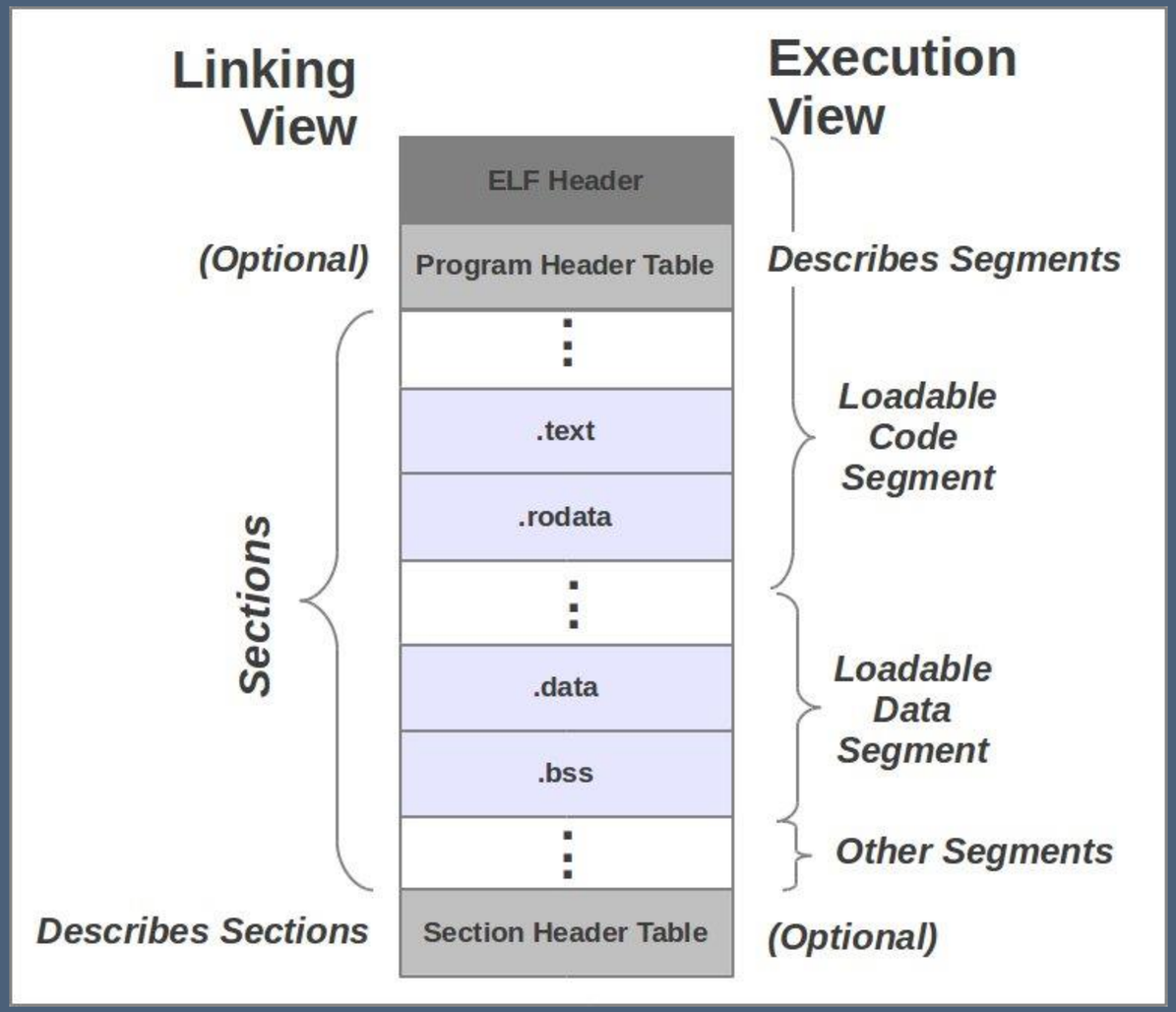
ELF Format

Layout of an ELF binary on disk



ELF Format

Program Headers vs. Sections



Sections versus Segments

An area that is often confusing when learning the ELF binary format

- Segments are modules of code that are loaded into memory on execution, defined in the program header table as type PT_LOAD
- *Sections* are logical areas of the code that don’t affect which code is mapped into memory, but some sections play a role in dynamic linking and must be there.
- As an example, relocatable objects (.o) do not contain any segments are described exclusively like sections. These can be utilized by the compile time linker to build the final ELF executable

```
ubuntu@ip-10-0-0-134:~$ readelf -lW /bin/ls

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x6aa0
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  PHDR           0x000040 0x0000000000000040 0x0000000000000040 0x0002d8 0x0002d8 R   0x8
  INTERP         0x000318 0x0000000000000318 0x0000000000000318 0x00001c 0x00001c R   0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000000000 0x0000000000000000 0x003458 0x003458 R   0x1000
  LOAD           0x004000 0x0000000000004000 0x0000000000004000 0x013091 0x013091 R E 0x1000
  LOAD           0x018000 0x0000000000018000 0x0000000000018000 0x007458 0x007458 R   0x1000
  LOAD           0x01ffd0 0x0000000000020fd0 0x0000000000020fd0 0x0012a8 0x002570 RW  0x1000
  DYNAMIC        0x020a58 0x0000000000021a58 0x0000000000021a58 0x000200 0x000200 RW  0x8
  NOTE           0x000338 0x0000000000000338 0x0000000000000338 0x000030 0x000030 R   0x8
  NOTE           0x000368 0x0000000000000368 0x0000000000000368 0x000044 0x000044 R   0x4
  GNU_PROPERTY   0x000338 0x0000000000000338 0x0000000000000338 0x000030 0x000030 R   0x8
  GNU_EH_FRAME   0x01cdcc 0x000000000001cdcc 0x000000000001cdcc 0x00056c 0x00056c R   0x4
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x10
  GNU_RELRO      0x01ffd0 0x0000000000020fd0 0x0000000000020fd0 0x001030 0x001030 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
   03     .init .plt .plt.got .plt.sec .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .data.rel.ro .dynamic .got .data .bss
   06     .dynamic
   07     .note.gnu.property
   08     .note.gnu.build-id .note.ABI-tag
   09     .note.gnu.property
   10     .eh_frame_hdr
   11
   12     .init_array .fini_array .data.rel.ro .dynamic .got
ubuntu@ip-10-0-0-134:~$
```

ELF Format

ELF Types

- Dynamically Linked (**ET_DYN**)
  - Shared object
  - An executable that links with shared objects at runtime
  - Is now the most commonly found type of ELF binary over static ELFs
- Statically Linked (**ET_EXEC**)
  - Linked at compile time: self-sufficient. Does not have ASLR, not a Position Independent Executable.
- Relocatable Object (**ET_REL**)
  - Building block
  - Kernel modules

ELF Format

Program Interpreter

- The dynamic runtime linker is the program interpreter for dynamically linked ELFs.
- PT_INTERP is a program header that specifies a path to the program interpreter.

[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]

Dynamic Linking

PLT/GOT

- Procedure Linkage Table (PLT) is a series of stubs that redirect execution to dynamically linked functions.
- Global Offset Table (GOT) is an array of dynamically linked function pointers.

```
0000000000001020 <puts@plt-0x10>:
    1020:       ff 35 ca 2f 00 00       push   QWORD PTR [rip+0x2fca]        # 3ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       ff 25 cc 2f 00 00       jmp    QWORD PTR [rip+0x2fcc]        # 3ff8 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]

0000000000001030 <puts@plt>:
    1030:       ff 25 ca 2f 00 00       jmp    QWORD PTR [rip+0x2fca]        # 4000 <puts@GLIBC_2.2.5>
    1036:       68 00 00 00 00          push   0x0
    103b:       e9 e0 ff ff ff          jmp    1020 <_init+0x20>
```

Dynamic Linking

PLT/GOT

- PLT stubs vary based on binding type
- Lazy vs. Eager binding

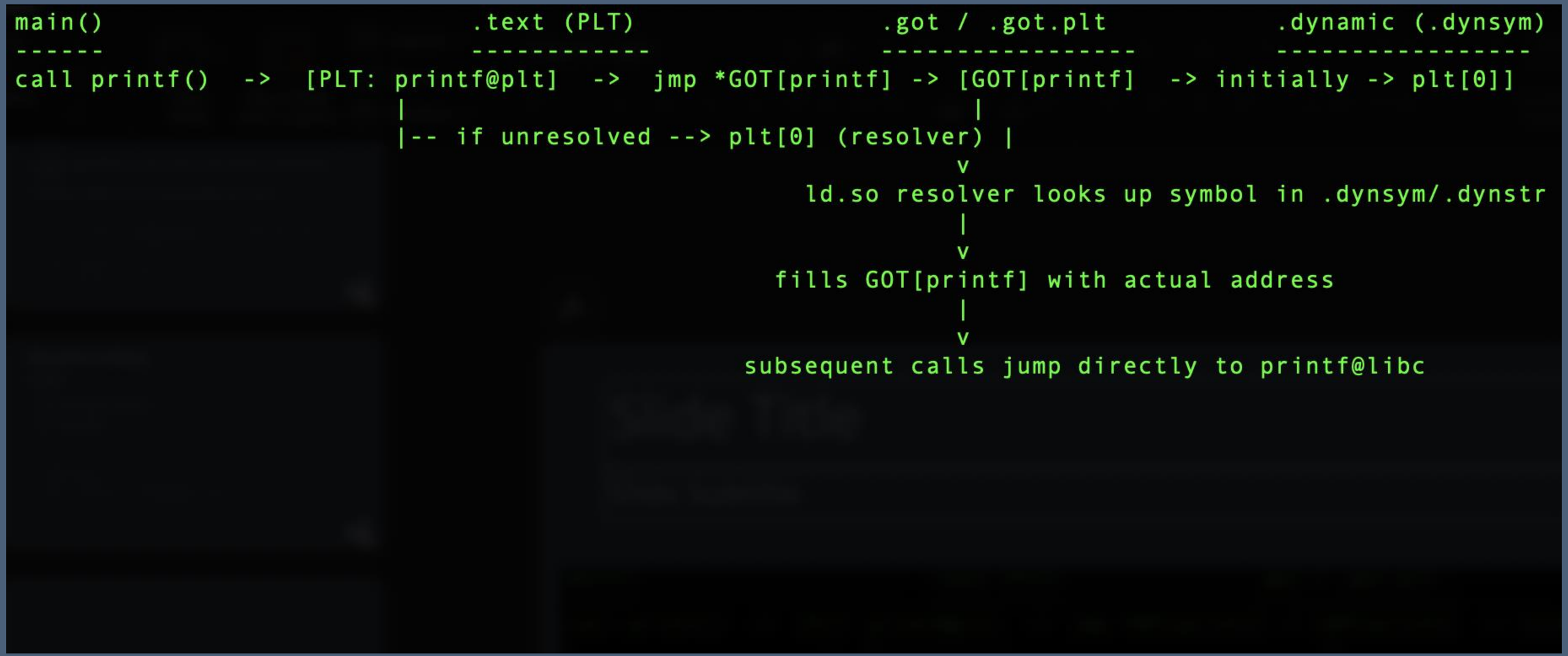
```
Disassembly of section .plt.sec:

0000000000001080 <puts@plt>:
    1080:       f3 0f 1e fa                 endbr64
    1084:       ff 25 2e 2f 00 00       jmp    QWORD PTR [rip+0x2f2e]        # 3fb8 <puts@GLIBC_2.2.5>
    108a:       66 0f 1f 44 00 00       nop    WORD PTR [rax+rax*1+0x0]
```

Dynamic Linking

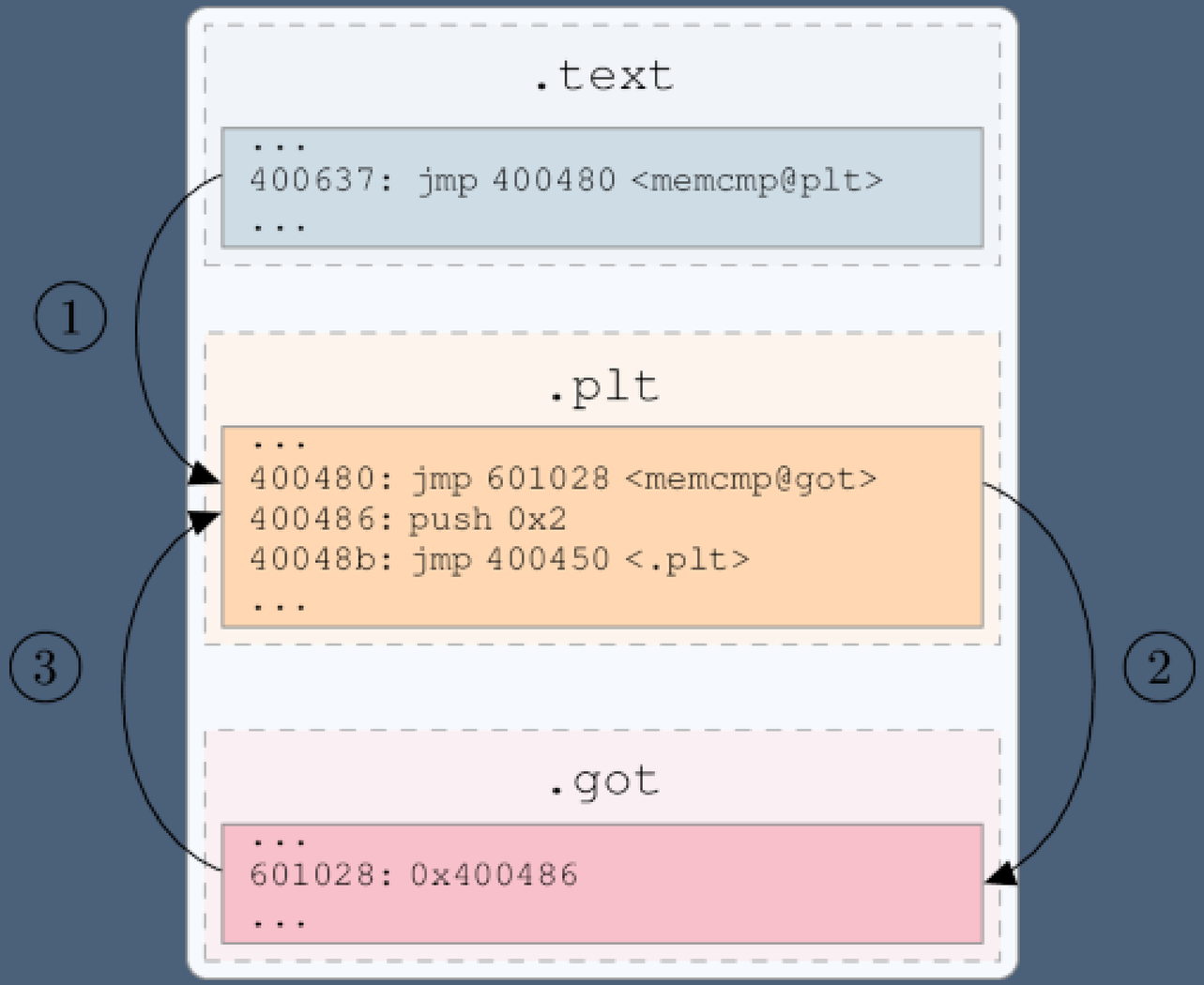
PLT/GOT

From main() → printf@plt (label: call printf@plt).
From printf@plt → GOT[printf] (label: jmp *GOT[printf]).
From GOT[printf] → plt[0] (if unresolved; label: initially → plt[0]).
From plt[0] → ld.so (label: resolver: push reloc index).
From ld.so → .dynsym (label: lookup symbol).
From ld.so → libc printf (label: resolve address).
From libc printf → GOT[printf] (label: write address).
subsequent calls go directly: printf@plt → GOT[printf] → printf@libc.


main()
.text (PLT)
.got / .got.plt
.dynamic (.dynsym)
call printf() -> [PLT: printf@plt] -> jmp *GOT[printf] -> [GOT[printf] -> initially -> plt[0]]
|-- if unresolved --> plt[0] (resolver)
ld.so resolver looks up symbol in .dynsym/.dynstr
fills GOT[printf] with actual address
subsequent calls jump directly to printf@libc

Dynamic Linking

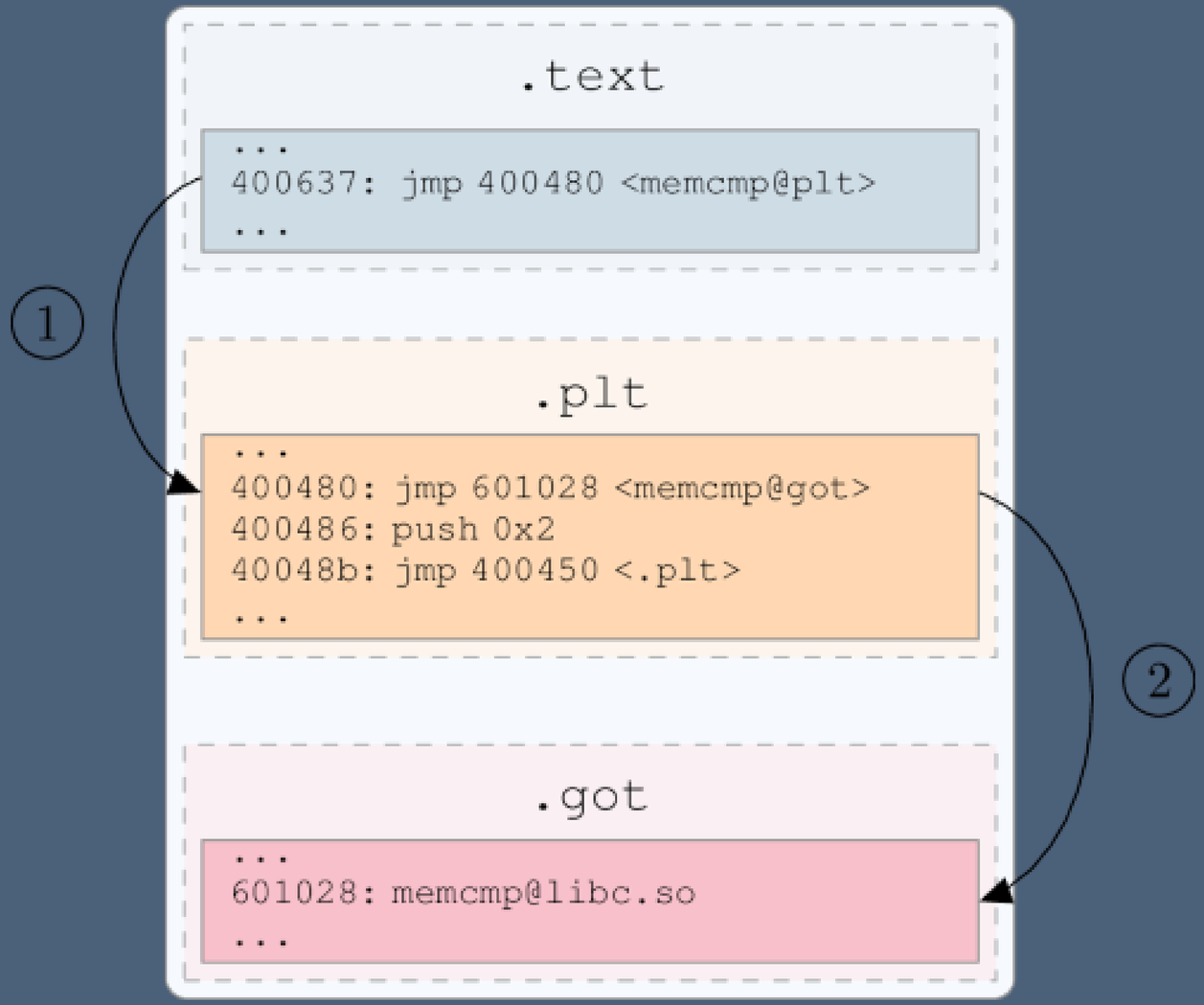
PLT/GOT



Credit: Romain Thomas (LIEF)

Dynamic Linking

PLT/GOT



Credit: Romain Thomas (LIEF)

Virus Requirements

- The host must execute our code and also execute the original host code to be considered an infector and not an overwriter
  - Hijacking Techniques
- We need room for our code
  - Infection Techniques
- The infection must not unintentionally break the host

History of ELF viruses

INFECTION

Infection Techniques

Shell infectors

Overwriters

Code caves

Forward text infection

Reverse text infection

PT_NOTE to PT_LOAD infection

.ctors / .dtors overwrite

Userland exec appender mmap/mprotect

got/.plt hijacking

LD_PRELOAD

Dynamic symbol hijacking – you can modify an R386 copy relocation into a specific location - there are global variables and functions

Preloading an entire linker yourself (Shiva) - https://tmpout.sh/2/6.html

Thread injection – Allows you to remotely inject a thread into a process that runs a hidden program within the program (Sarumon)

Infection Techniques

Silvio Cesare Text Padding

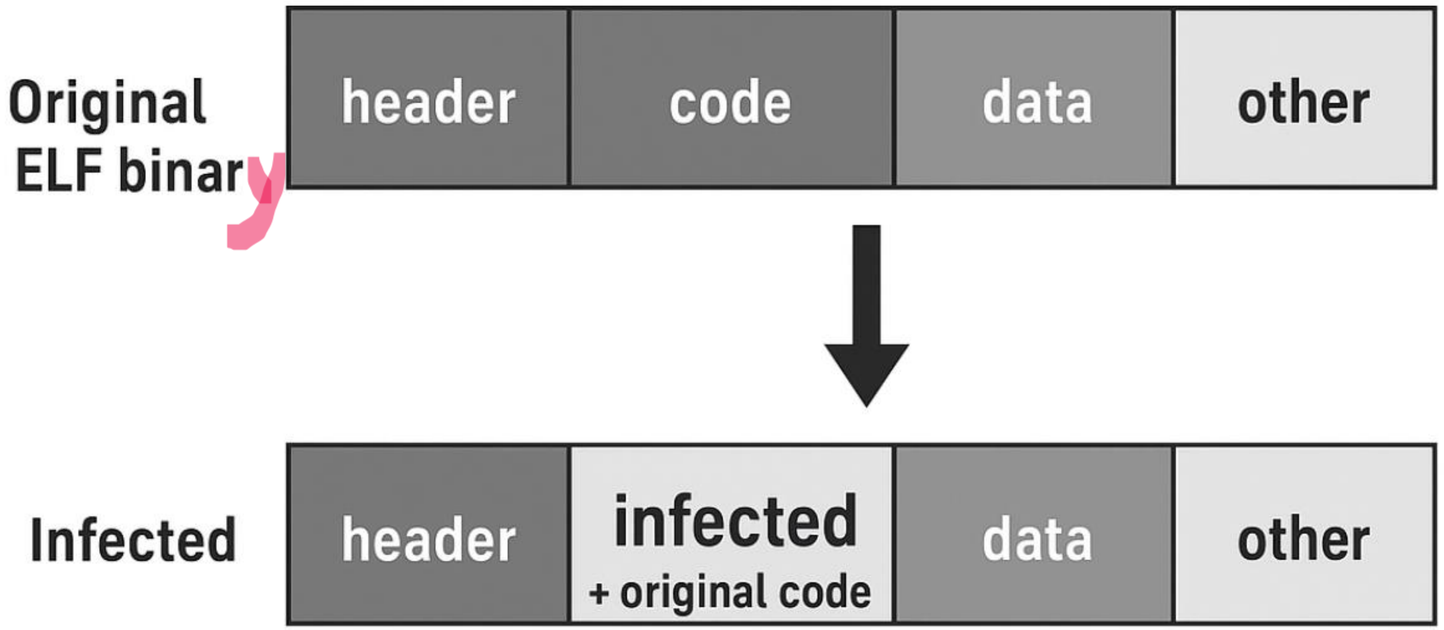
- Recall that the code segment of the ELF is padded to page end.
- Placing our parasite in the padding requires little changes to the ELF:
  - Minimal changes to segments that follow the code segment - no need to fixup the data references in code.
- Described by Silvio Cesare in his UNIX VIRUSES paper in the 90s.
- Other text infection techniques
  - Reverse Text Padding technique (Linux.Nasty by TMZ, Skeksi by elfmaster)
  - SCOP Ultimate Text Infection by elfmaster

Infection Techniques

Reverse Text Segment Infection

- The **Reverse Text Segment Infection** technique works by retroactively creating space at the beginning of an ELF executable file. The core step involves manipulating the **Program Header Table (PHT)** to decrease both the **virtual address (p_vaddr)** and the **file offset (p_offset)** of the .text segment, which effectively shifts the entire segment backward in the file and memory to carve out a new gap before the original code. The **parasite code** is then inserted into this gap, and the **ELF header's entry point** is modified to point to the start of the injected virus code. After the malicious code executes, it ensures program integrity by jumping to the host's original entry point, allowing the host program to run normally, while the segment's size fields (p_filesz and p_memsz) are updated to encompass the new, enlarged region.

REVERSE TEXT SEGMENT INFECTION



Infection Techniques

PT_NOTE -> PT_LOAD

- Loadable segment (**PT_LOAD**) is the most common type: code and data segments are loadable segments.
- Note segment (**PT_NOTE**) contains metadata, normally not required for execution.
- Modifying the program header table, we can turn a note segment into a loadable segment with RX permissions and point it to the end of file.

Infection Techniques

Code Replacement

- Replacing host’s code.
- Statically linked ELF’s often contain multiple versions of the same function, e.g. memcpy implementations that utilize optimized instructions (Credit: sad0p).
- Overwriting code that is used by the host, but absence of which doesn’t affect functionality, e.g. debug logging.
- Not very portable.

Infection Techniques

.ctors / .dtors overwrite

- the **.ctors** and **.dtors** sections hold pointers to constructor and destructor functions that the runtime calls automatically before main() and after program exit; an overwrite-style infection leverages that predictable execution path by replacing or redirecting those pointers so attacker code runs as part of the program's normal startup/shutdown sequence. Conceptually the malicious actor doesn't need to alter the program entry point or insert complex trampolines — they simply tamper with the list of function pointers so their payload is invoked early, and (in more cautious variants) try to preserve a pointer to the original function so normal program behaviour continues.

Infection Techniques

Userland exec appender mmap/mprotect

- This technique allocates a fresh region of memory inside the process at runtime (usually mmap() and mprotect()), changes that region’s protections so it is executable, places the virus payload into that region, and then transfers control into it; once the payload finishes, control is passed back to the program’s original entry point so the host appears to run normally. Because the malicious code executes from a dynamically created, in-memory page rather than by permanently modifying on-disk code sections, the on-disk binary can look unchanged while the in-memory image is altered at startup — a property that can make detection harder.

HIJACKING

Hijacking

Entry-point hijacking

- Pros
  - Easy to implement
- Cons
  - Boring
  - Easy to detect
- Solution: Entry Pint Obscuring (EPO)

Hijacking

EPO: Constructor/destructor hijacking

- The .init_array and .fini_array are function pointer arrays that point respectively to constructor and destructor routines.
- Overwriting an entry in one of those arrays is a quick and effective technique.
- Must still pass execution to the original constructor/destructor.
- Program may exit before it calls the destructors.
- Lin64.Eng3ls written by s01den for tmp.out #1 uses .init_array to achieve EPO.

Hijacking

EPO: PLT Hijacking

- My virus Linux.ElizaCanFix (tmp.out #3) hijacks the __cxa_finalize PLT entry to achieve EPO. This is an alternative to destructor hijacking.

```
0000000000001040 <__cxa_finalize@plt>:
    1040:       ff 25 7e 2f 00 00       jmp    QWORD PTR [rip+0x2f7e]        #  <__cxa_finalize@GLIBC_2.2.5>

            ... after infection ...

0000000000001040 <__cxa_finalize@plt>:
    1040:       e9 44 01 00 00          jmp    1189 <_fini+0x9>
```

Hijacking

EPO: Relocation hijacking

- A technique described by sad0p in Black Mass II.
- This technique achieves hijacking by modifying the target function address in the dynamic symbol table. When the host code calls a legitimate function, the execution is redirected to the virus.
- Virus still must save the state and pass execution to the original function.
- Depending on the hijacked function, the virus may be never called or called multiple times.

Hijacking

EPO: Inline Hijacking

- Patching call sites directly
  - Call offset patching
  - Function pointer argument patching

Hijacking

EPO: Inline Hijacking

- My virus Linux.Slotmachine hijacking execution in the _start stub:

```
000000000000006c0 <_start>:
                ...
 6d0:   f94003e1        ldr     x1, [sp]
 6d4:   910023e2        add     x2, sp, #0x8
 6d8:   910003e6        mov     x6, sp
 6dc:   f00000e0        adrp    x0, 1f000 <__FRAME_END__+0x1e6d4>
 6e0:   f947ec00        ldr     x0, [x0, #4056]
 6e4:   d2800003        mov     x3, #0x0                        // #0
 6e8:   d2800004        mov     x4, #0x0                        // #0
 6ec:   97ffffe1        bl      670 <__libc_start_main@plt>
```

```
000000000000006c0 <_start>:
                ...
 6d0:   f94003e1        ldr     x1, [sp]
 6d4:   910023e2        add     x2, sp, #0x8
 6d8:   910003e6        mov     x6, sp
 6dc:   90000180        adrp    x0, 30000 <__bss_end__+0xffc0>
 6e0:   d503201f        nop
 6e4:   d2800003        mov     x3, #0x0                        // #0
 6e8:   d2800004        mov     x4, #0x0                        // #0
 6ec:   97ffffe1        bl      670 <__libc_start_main@plt>
```

Hijacking

Pre-Loading: Shared Object

- Shared object pre-loading is a legitimate mechanism, used in debugging, profiling, and patching.
- Pre-loaded shared objects can be specified through the LD_PRELOAD environment variable or modifying the /etc/ld.so.preload configuration file.
- Conveniently, it allows attackers to intercept library calls for fun purposes.
- Used in userland “rootkits” for hijacking calls to libc functions.
- Kinda lame, also not a virus…

- Injecting shared objects through dynamic table modification is more interesting.

- Injecting malicious code via **Dynamic Table Modification (DTM)**, specifically targeting the **.dynamic section** of an ELF binary, is a stealthy method for gaining execution control without modifying the .text segment's code. This technique exploits the **dynamic linker's** functionality by adding a new entry to the host binary's DT_NEEDED list in the .dynamic table. This newly added entry points to an attacker-controlled **shared object (library)** containing the malicious code. When the host program is executed, the operating system's dynamic linker automatically resolves and loads this malicious shared object into the process memory, granting execution to its constructor function (_init) before the host's legitimate code runs, thus achieving a persistent, file-based infection.

Hijacking

Pre-Loading: Linker

- Described by elfmaster in tmp.out #2.
- Modification of program interpreter path (**PT_INTERP**) to a custom runtime linker.
- This custom linker will load the virus object as well as the system runtime linker into memory.
- Can optionally hijack the runtime linking process to implement LD_PRELOAD style debauchery.

Propagation

- We must vet the target before infection
  - Is it a valid ELF?
  - Can the target support me?
  - Will the target crash if I run?

ADVANCED

Advanced

Obfuscation: Packing and Encryption

- The virus payload is encrypted and optionally compressed before infecting the target.
- On disk, it commonly appears as a high entropy blob and a decryption routine.
- An early attempt to fight detection through signatures… so anti malware solutions would just create signatures of the decryptor!

Advanced

Mutation

- First came on the scene in the late 80s/early 90s as virus authors looked for new ways to bypass early antivirus solutions.
- With each propagation event, the virus evolves through self-modification.

Advanced

Mutation: Oligomorphism

- Oligomorphism - ολίγος (oligos) “few” + μορφή (morphe) “shapes”.
- Oligomorphic viruses are capable of building a finite number of decryption mechanisms with each generation.
- The decrypted payload remains the same.

Advanced

Mutation: Polymorphism

- Polymorphism - πολύ (poly) “many" + μορφή (morphe) “shapes”.
- Polymorphic viruses are capable of building an infinite number of decryption mechanisms with each generation.
- The decrypted payload remains the same.

Advanced

Mutation: Metamorphism

- Metamorphism - μετά (meta) “change” + μορφή (morphe) “shapes”.
- Metamorphic viruses are more sophisticated, as the payload itself is modified with each generation.
  - Garbage instructions
  - Instructions replaced with equivalents
  - Code block shuffling
  - Local compiler/assembler utilization
  - Many more examples!

Advanced

Other

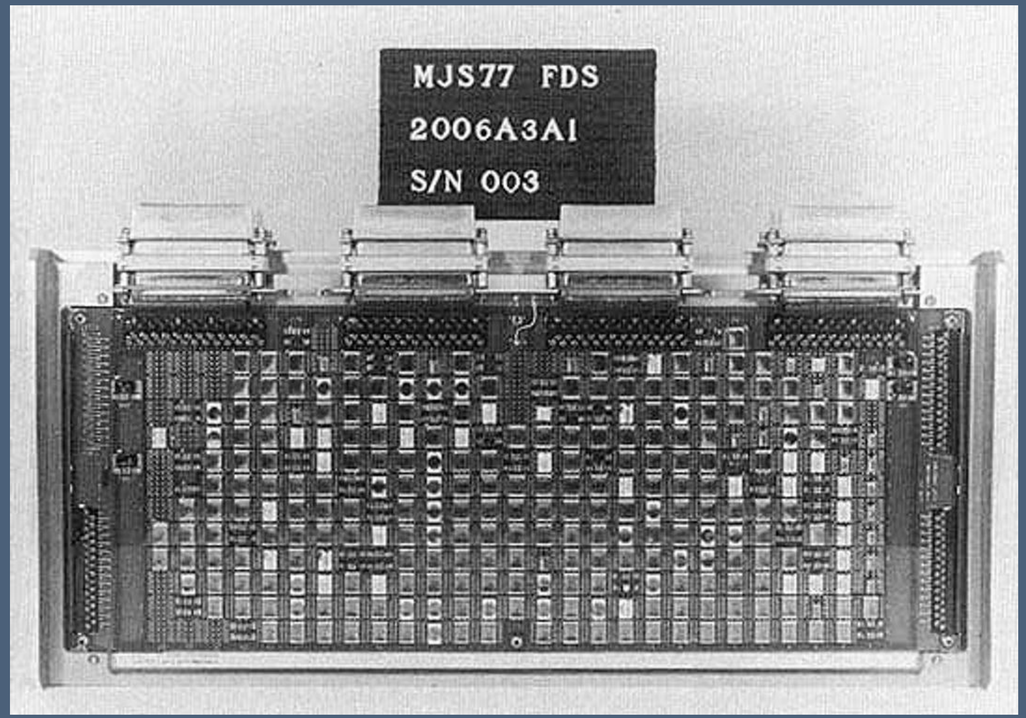
- VM obfuscation: Lin64.M4rx virus by s01den (tmp.out #2)
- JVM Virus: Java.Cheshire.a by b0t
- Immediate value obscuring through system registers: Linux.Slotmachine (tmp.out #4)

Other ELF shenanigans

- Code golf
- Switching the Endianess byte
- Don’t miss spooky’s talk on Wednesday!

Interstellar hacking

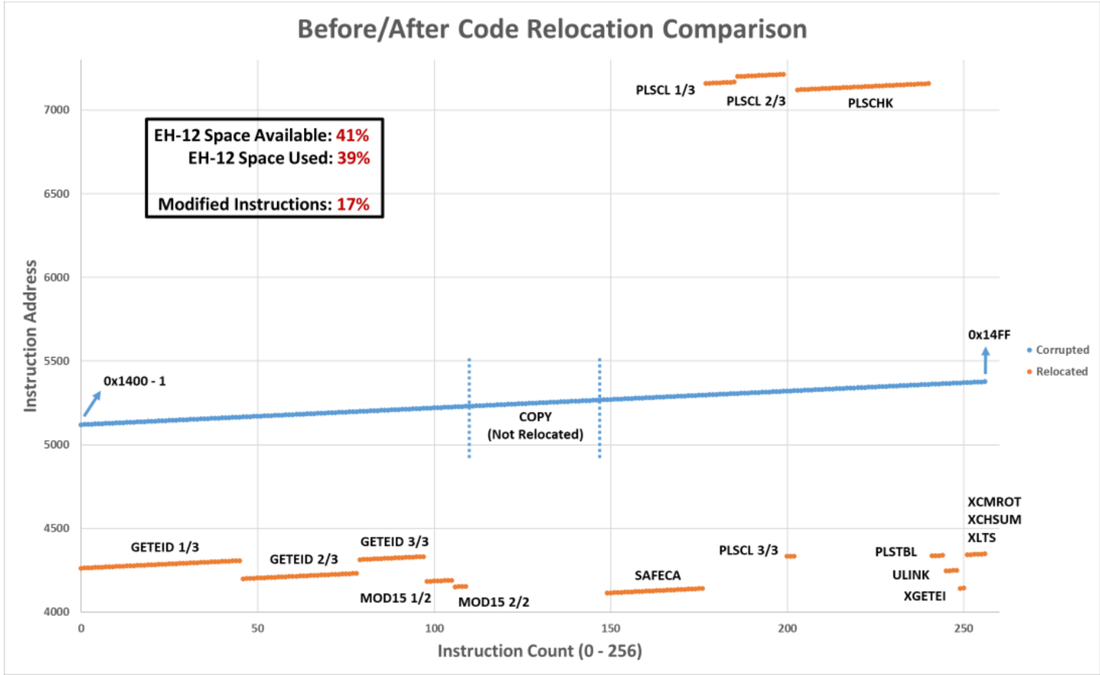
- In 2023 the Flight Data Subsystem on Voyager-1 has experienced a failure: 256 16-bit word segment of the RAM has become unusable.
- Code only exists in RAM and has been running there for decades.
- Every 2.5 ms an interrupt restarts execution.
- P Tables (subroutine jump tables) are used to select the target subroutine. P Counter tracks the next routine to execute.



Interstellar hacking

- Solution
  - Find 256 words worth of code caves
  - Relocate all of the affected code
  - Fixup all code references
- To test the solution, JPL engineers injected a CMROT program. First interstellar computer virus?

Statistics (thanks to Armen Arslanian)



Resources

- https://tmpout.sh
- https://binary.golf
- Learning Linux Binary Analysis - Ryan O’Neill (elfmaster)

QUESTIONS