O:11:"WikiContent":16:{s:8:"fmt_html";s:7489:"

Contents (hide)

1 Introduction
2 From the Specification
3 Virtual Memory
4 The Linker's Job
5 The Static Loader

Lab 9 (ASM) - Reading Materialshow_sections_edit()){?>name,"edit","action=edit&section=1")?>

Introductionshow_sections_edit()){?>name,"edit","action=edit&section=2")?>

In this lab you will create a simple virus. Your virus will only need to be able to infect very simple ELF executables. Naturally, you will need to write it in strictly position-independent code. You will need to learn to manipulate the ELF header and the program headers in an ELF executable, make sure you fully understand these parts of the ELF specification. You will be using Linux system calls for all access to the files, make sure you know how to use them: open, read, write, lseek, close, and exit. In order to simplify your task we have provided some skeleton code with macros for calling these system services (see task 0b).

From the Specificationshow_sections_edit()){?>name,"edit","action=edit&section=3")?>

Program headers and static program loading are described in pages 2-1 up to 2-9 of the .

Some additional information is given below.

Virtual Memoryshow_sections_edit()){?>name,"edit","action=edit&section=4")?>

Modern operating systems employ a scheme called Virtual Memory. This scheme enables each process to have its own view of memory, independent of other processes. The operating system (with help from the hardware) maps process memory (virtual pages) into real memory (real pages).

This way, each process can pretend it is the only process running on the system, and trust the operating system to ensure its memory does not collide with other processes.

The Linker's Jobshow_sections_edit()){?>name,"edit","action=edit&section=5")?>

The introduction of virtual memory makes life much easier on the compiler and linker. The compiler generates code which thinks it is located at the beginning of memory (address 0x0), and leaves information for the linker on where corrections need be made. The linker is in charge of choosing the memory layout of the program, and can decide to which address in virtual memory the code will be loaded to.

A simple example will help illustrate the point. Look at the following code:

```
char *message = "hello linker";

void foo(){
    printf("%s\n",message);
}
```

Compiling this code to produce an object file ends up looking like this:

When the linker is asked to link this code, and make it an executable, it needs to decide on the memory layout first: what will the virtual address of `foo()` will be, and where in virtual memory will `message` be located?

After deciding on the memory layout, the linker needs to inform the loader how to load the executable. This is done by using program headers in the ELF format.

The Static Loadershow_sections_edit()){?>name,"edit","action=edit&section=6")?>

The loader runs when the exec system call is invoked, i.e. whenever an executable file is run. The job of the Loader is to load the executable into main memory. It does so by reading the program headers located in the ELF formatted executable, and acting accordingly. Your virus should change a program header in the ELF executable file it infects so as to make the loader load your virus code when the infected ELF executable is run. Let us take a look at the program header structure in an ELF file:

```
typedef struct {
        Elf32_Word      p_type;         /* entry type */
        Elf32_Off       p_offset;       /* file offset */
        Elf32_Addr      p_vaddr;        /* virtual address */
        Elf32_Addr      p_paddr;        /* physical address */
        Elf32_Word      p_filesz;       /* file size */
        Elf32_Word      p_memsz;        /* memory size */
        Elf32_Word      p_flags;        /* entry flags */
        Elf32_Word      p_align;        /* memory/file alignment */
} Elf32_Phdr;
```

- `p_type`: The type of the entry. We are only interested in PT_LOAD, which means the loader must load the appropriate data from the file into memory.
- `p_offset`: The offset in the file, from which we start to load data.
- `p_vaddr`: The virtual address to which we load the data.
- `p_paddr`: The physical address. On x86 we can safely ignore this.
- `p_filesz`: Total amount of data which need to be mapped from the file.
- `p_memsz`: Total amount of data which needs to be mapped (can differ from `p_filesz`).
- `p_flags`: The flags:
  - PF_R: map for reading
  - PF_w: map for writing
  - PF_X: map for execution
- `p_align`: The alignment needed. The linker must make sure this section's virtual address equals 0 module p_align.

One remarks is in order: `p_filesz` can be different from `p_memsz`. This can happen when, for example, we need to allocate space for uninitialized variables in memory. There is no point in wasting space in the executable file for such variables (the section which holds these variables is traditionally called the ".bss" section). But in this lab, `p_filesz` should be the same as `p_memsz`.

Use picture, which illustrates the ELF format of executable files.

Make sure that you understand picture, which describes the structure of the memory when infected file is run. ";s:4:"diff";a:5:{i:0;a:4:{s:4:"date";i:1497784897;s:4:"diff";s:110:"17c17 < [[file:lab7_elf.pdf|ELF file specification]]. --- > [[file:ELF_Format.pdf|ELF file specification]]. ";s:6:"author";s:6:"llutan";s:12:"edit-summary";s:0:"";}i:1;a:4:{s:4:"date";i:1496902588;s:4:"diff";s:79:"53c53 < [[file:object.jpg|object.jpg]] --- > [[image:object.jpg|object.jpg]] ";s:6:"author";s:7:"tamirgr";s:12:"edit-summary";s:0:"";}i:2;a:4:{s:4:"date";i:1496902557;s:4:"diff";s:73:"53c53 < [[image:lab9/object.jpg]] --- > [[file:object.jpg|object.jpg]] ";s:6:"author";s:7:"tamirgr";s:12:"edit-summary";s:0:"";}i:3;a:4:{s:4:"date";i:1486655900;s:4:"diff";s:510:"108c108 < Use [[file:lab9/Chart1.png|this]] picture, which illustrates the ELF format of executable files. --- > Use [[file:lab9_Chart1.png|this]] picture, which illustrates the ELF format of executable files. 112c112 < Make sure that you understand [[file:lab9/Chart2.png|this]] picture, which describes the structure of the memory when infected file is run. --- > Make sure that you understand [[file:lab9_Chart2.png|this]] picture, which describes the structure of the memory when infected file is run. ";s:6:"author";s:2:"oa";s:12:"edit-summary";s:0:"";}i:4;a:4:{s:4:"date";i:1486655872;s:4:"diff";s:108:"17c17 < [[file:lab7/elf.pdf|ELF file specification]]. --- > [[file:lab7_elf.pdf|ELF file specification]].

";s:6:"author";s:2:"oa";s:12:"edit-summary";s:0:"";}}s:4:"name";s:22:"Lab92.Reading_Material";s:3:"fmt";s:5375:"

Lab 9 (ASM) - Reading Material

Introduction

In this lab you will create a simple virus. Your virus will only need to be able to infect very simple ELF executables. Naturally, you will need to write it in strictly position-independent code. You will need to learn to manipulate the ELF header and the program headers in an ELF executable, make sure you fully understand these parts of the ELF specification. You will be using Linux system calls for all access to the files, make sure you know how to use them: open, read, write, lseek, close, and exit. In order to simplify your task we have provided some skeleton code with macros for calling these system services (see task 0b).

From the Specification

Program headers and static program loading are described in pages 2-1 up to 2-9 of the [[file:ELF_Format.pdf|ELF file specification]].

Some additional information is given below.

Virtual Memory

Modern operating systems employ a scheme called Virtual Memory. This scheme enables each process to have its own view of memory, independent of other processes. The operating system (with help from the hardware) maps process memory (virtual pages) into real memory (real pages).

This way, each process can pretend it is the only process running on the system, and trust the operating system to ensure its memory does not collide with other processes.

The Linker's Job

The introduction of virtual memory makes life much easier on the compiler and linker. The compiler generates code which thinks it is located at the beginning of memory (address 0x0), and leaves information for the linker on where corrections need be made. The linker is in charge of choosing the memory layout of the program, and can decide to which address in virtual memory the code will be loaded to.

A simple example will help illustrate the point. Look at the following code: `char *message = "hello linker"; void foo(){ printf("%s\n",message); }` Compiling this code to

produce an object file ends up looking like this:
[[image:object.jpg|object.jpg]]

When the linker is asked to link this code, and make it an executable, it needs to decide on the memory layout first: what will the virtual address of `foo()` will be, and where in virtual memory will `message` be located?

After deciding on the memory layout, the linker needs to inform the loader how to load the executable. This is done by using program headers in the ELF format.

The Static Loader

The loader runs when the exec system call is invoked, i.e. whenever an executable file is run. The job of the Loader is to load the executable into main memory. It does so by reading the program headers located in the ELF formatted executable, and acting accordingly. Your virus should change a program header in the ELF executable file it infects so as to make the loader load your virus code when the infected ELF executable is run. Let us take a look at the program header structure in an ELF file: `typedef struct { Elf32_Word p_type; /* entry type */ Elf32_Off p_offset; /* file offset */ Elf32_Addr p_vaddr; /* virtual address */ Elf32_Addr p_paddr; /* physical address */ Elf32_Word p_filesz; /* file size */ Elf32_Word p_memsz; /* memory size */ Elf32_Word p_flags; /* entry flags */ Elf32_Word p_align; /* memory/file alignment */ } Elf32_Phdr;` (*>) (*) `p_type`: The type of the entry. We are only interested in PT_LOAD, which means the loader must load the appropriate data from the file into memory. (*) `p_offset`: The offset in the file, from which we start to load data. (*) `p_vaddr`: The virtual address to which we load the data. (*) `p_paddr`: The physical address. On x86 we can safely ignore this. (*) `p_filesz`: Total amount of data which need to be mapped from the file. (*) `p_memsz`: Total amount of data which needs to be mapped (can differ from `p_filesz`). (*) `p_flags`: The flags: (*>) (*) PF_R: map for reading (*) PF_w: map for writing (*) PF_X: map for execution ( `p_align`: The alignment needed. The linker must make sure this section's virtual address equals 0 module p_align. (

One remarks is in order: `p_filesz` can be different from `p_memsz`. This can happen when, for example, we need to allocate space for uninitialized variables in memory. There is no point in wasting space in the executable file for such variables (the section which holds these variables is traditionally called the ".bss" section). But in this lab, `p_filesz` should be the same as `p_memsz`. Use [[file:lab9_Chart1.png|this]] picture, which illustrates the ELF format of executable files.

Make sure that you understand [[file:lab9_Chart2.png|this]] picture, which describes the structure of the memory when infected file is run.
";s:6:"author";s:6:"llutan";s:7:"created";i:1486381056;s:13:"last_modified";i:1497784898;s:11:"description";s:0:"";s:8:"keywords";a:0:{}s:16:"content_keywords";a:20:{s:6:"memory";i:17;s:4:"file";i:12;s:10:"executable";i:11;s:3:"elf";i:11;s:7:"virtual";i:9;s:4:"need";i:9;s:7:"program";i:8;s:4:"code";i:8;s:4:"make";i:7;s:6:"header";i:7;s:6:"linker";i:7;s:7:"address";i:6;s:4:"load";i:6;s:6:"loader";i:6;s:7:"process";i:6;s:4:"data";i:5;s:4:"viru";i:4;s:8:"p_filesz";i:4;s:4:"sure";i:4;s:7:"reading";i:4;}s:4:"tags";a:0:{}s:8:"comments";s:0:"";s:6:"hidden";b:0;s:5:"allow";a:0:{}s:6:"secure";b:0;s:13:"redirect_page";s:0:"";}