TU WIEN Informatics

Emulating a.out executables in Linux user space

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Siegfried Oleg Pammer
Registration Number 01633095

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Anton Ertl

Vienna, 1st May, 2021

Siegfried Oleg Pammer	M. Anton Ertl

Erklärung zur Verfassung der Arbeit

Siegfried Oleg Pammer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Mai 2021

Siegfried Oleg Pammer

Danksagung

Mein hauptsächlicher Dank gilt meinem Betreuer M. Anton Ertl, für die Unterstützung und Begleitung während dieser Arbeit.

Dank gebührt außerdem Daniel Grunwald für die hilfreichen Kommentare zu meiner Arbeit, und der ic#code Gruppe für die Erfahrungen, die ich in den letzten 13 Jahren machen konnte, und dass sie mein Interesse an Übersetzerbau und verwandten Themen geweckt haben.

Acknowledgements

Mainly I want to thank my advisor M. Anton Ertl for his support during the writing of this thesis.

I want to also thank Daniel Grunwald for his helpful comments, and the ic#code group for the experience I could gain during the last 13 years, and that they sparked my interest in compiler construction and related topics.

Kurzfassung

Die Einstellung des Supports und die Löschung des `binfmt_aout` Kernelmoduls aus dem Linux Kernel in der Version 5 machen es unmöglich, Programme auszuführen, die das `a.out` Format verwenden. Diese Arbeit versucht die Frage zu beantworten, ob und wie es möglich ist, `a.out` Binärdateien auf modernen Systemen auszuführen. Ein Hauptziel war, dass die Lösung im „user space“ ausgeführt wird, ohne besonderen Support durch den Kernel. In dieser Arbeit, befassen wir uns zunächst mit den Problemen, die auftreten, wenn man versucht `a.out` Binärdateien zu laden, weil sich Plattform für die `a.out` ursprünglich entwickelt wurde, in manchen Aspekten sehr von modernen Plattformen unterscheidet. Dann führen wir mögliche Lösungen für diese Probleme an und präsentieren auch einen funktionalen Prototyp, der die Ausführung von `a.out` Programmen auf aktuellen Linux Kerneln/Distributionen ermöglicht.

Abstract

The depreciation and removal of the `binfmt_aout` kernel module, in version 5 of the Linux Kernel, makes it impossible to run executables, compiled using this old format. This work tries to answer the question whether it is possible to execute `a.out` binaries on modern systems, and how support for the `a.out` format can be implemented. One of the main goals was that the solution should be running in user space, i.e., without special kernel support. In this work, we first enumerate the problems which crop up when trying to load `a.out` binaries, as the platforms on which they were original developed are very different in some aspects. Then we present possible solutions to each of these problems and also a full working solution that allows us to execute `a.out` programs on current Linux kernels/distributions.

Contents

CHAPTER 1

Introduction

In March of 2019 Linus Torvalds published a change [17] [18] to the Linux kernel, in which support for the `a.out` format was deprecated and subsequently removed (this went into effect as of kernel version `5.0`). The reason for this (as described in [22]) is that `a.out` "coredumping has bitrotten quite significantly and would need some fixing to get it into shape again". However, as there are still people interested in being able to run `a.out` binaries on current versions of Linux, this work aims to find a solution that does not require special kernel support. Another reason for this work is that, although we were able to run select `a.out` QMAGIC binaries on recent 32-bit Linux systems (Ubuntu 16.04 LTS 32-bit) using the `binfmt_aout` kernel module, it turned out to be very brittle and somewhat difficult to configure. Also it was only possible to get to work QMAGIC binaries, but not all other variants of `a.out`.

This work will briefly discuss how operating systems generally load binaries using the example of `a.out` on Linux. Further it will explore the characteristics and uses of the variants of the `a.out` format, as well as the downsides, which led to `a.out` gradually being replaced by ELF around 1995 and finally deprecated and removed in 2019. Furthermore, this work will explore the options and work that would have to be done to allow Linux `a.out` binaries to run on current kernels and systems and finally present a possible implementation of such a loader.

CHAPTER 2

The Environment

In the following chapter the prerequisites of this work will be laid out. First, a short summary of the history of `a.out` binaries will be given, followed by an explanation of the `ptrace` API. For those interested, a short summary of the x86 architecture and Linux system calls may be found in the appendix (see 7.2 and 7.3).

2.1 A brief history of the `a.out` format

Modern operating systems usually support virtual memory and address spaces, which allows different processes to use the same (virtual) addresses independently. Likewise, many modern executable formats, including Executable and Linking Format (ELF)[9] used today on Linux, and Portable Executable (PE) used on Microsoft Windows, use multiple sections (sometimes called "segments"), with each section serving a specific purpose. Most common are the `.text` and `.data` sections, used for executable code and data bytes, respectively.

Splitting the executable into multiple sections has several advantages[16, page 50]: Sections that contain executable code can be marked executable and read-only when loaded by the operating system. This prevents modification and allows the operating system to reuse pages of memory that belong to the same library in multiple processes – saving considerable amounts of memory.

This development started with the PDP-11 which was first introduced in 1970. Because it used 16-bit addressing, it limited the user to a total of only 64KB. In order to work around this limitation, later iterations used virtual memory (PDP-11/40, 1972[5, page 6-2]) and separate address spaces (PDP-11/45, 1973) for code (called "I(nstruction) space") and data (called "D(ata) space")[6, page 145], enabling the use of 64KB of code and also 64KB of data per process. The various instructions operated on either the I or D space. Additionally programs could use Status Register #3 to decide whether

they want to use a "combined" address space or separate I and D spaces. The tools in use – compilers, assemblers and linkers – were modified to create two-section object files, leading to the birth of the `a.out` format[1].

2.2 The `ptrace` API

`ptrace` is a very powerful system call, which can be used to control execution of a process and examine that process's memory and registers. Many Linux tools, including `gdb` and `strace`, rely on `ptrace` to provide their functionality.

The general usage patterns are that the target process (the "tracee") is either launched by the "tracer" process or it the tracer process attaches itself to the tracee. The pattern used by this work falls into the first category.

The `ptrace` API provides one function that can perform a variety of tasks, including initialization of a trace, setting options, reading and writing registers and memory, listening for signals and system calls, and single-stepping instruction by instruction.

The function accepts four parameters: the request kind, the PID of the target process and two pointer values: address and data. The request kind describes, which action should be performed on the target process. The address and data values are used for transferring data to the target process. The return value either denotes the result of an operation or contains data retrieve from the process.

Note that `ptrace` only affects the process it is first attached to. The proof of concept discussed in this work only traces one process. If there are `a.out` programs to be executed that spawn multiple child processes, this may cause problems, if these processes try to load additional libraries.

The requests used in this work are the following:

- `PTRACE_TRACEME`: This request lets us indicate that the current process should be traced. The initialization pattern used in this work is the following: use `fork` to create a new process space, prepare tracing via the `PTRACE_TRACEME` request, then launch the actual tracee using `execve`.
- `PTRACE_SETOPTIONS`: This request allows the tracer to configure the trace. See 4.2 for more information on which options are used.
- `PTRACE_SINGLESTEP` lets the tracer execute single instructions in the tracee step by step and regain control afterwards.
- `PTRACE_CONT` lets the tracee continue and also allows to send signals to the tracee.

[1] `a.out` stands for "assembler output". Although the name "a.out" is still used today as the default output name when compiling a program with GCC, these output files do not use the `a.out` format.[13]

- PTRACE_SYSCALL: A combination of this request and waitpid is used to intercept system calls. After the request is made the tracee is stopped in the syscall enter phase and the tracer can examine registers before the system call. If the instruction pointer (in the EIP register) is modified by the tracer at this point, the system call can be cancelled and other code can be executed instead. The tracee may be resumed by using PTRACE_CONT or PTRACE_SINGLESTEP. If the tracer wishes to continue normally, this can be done by using PTRACE_SYSCALL, which simply continues execution until the system call exit phase. After that the tracer may examine the result of the system call.

- PTRACE_GETREGS is used to read the contents of all registers at the current point of execution. For this a pointer to a struct user_regs_struct (see listing 2.1) is passed as data pointer.

- PTRACE_SETREGS is used to write the contents of all registers at once. Again a pointer to a struct user_regs_struct is passed, which contains the register values that should be applied.

- PTRACE_PEEKDATA[2] is used to read a 64-bit value from the tracee's memory. The read value is returned.

- PTRACE_POKEDATA is used to write a 64-bit value to the tracee's memory.

Listing 2.1: Definition of the 32-bit version of the struct user_regs_struct for reference

```
1  struct user_regs_struct
2  {
3    long int ebx;            // EBX register
4    long int ecx;            // ECX register
5    long int edx;            // EDX register
6    long int esi;            // ESI register
7    long int edi;            // EDI register
8    long int ebp;            // EBP register
9    long int eax;            // EAX register: on system call
      enter, this holds the
10                            //               system call number;
       on system call
11                            //               exit, the holds the
       return value.
12   long int xds;            // data segment register
13   long int xes;            // extra segment register
```

[2]There are also PTRACE_PEEKTEXT and PTRACE_POKETEXT requests, respectively, however the documentation states that both kinds of requests can operate on all memory pages. Currently, there is no distinction between .text and .data.

```
14    long int xfs;            // extra segment register 2
15    long int xgs;            // extra segment register 3
16    long int orig_eax;       // orig. EAX register: on system
       call exit,
17                             //                     holds the
       orig. system call number.
18    long int eip;            // EIP register
19    long int xcs;            // code segment register
20    long int eflags;         // flags register
21    long int esp;            // ESP register
22    long int xss;            // stack segment register
23 };
```

2.3 The Loader

Before an executable is executed, usually a new process is created. Linux uses three different system calls to create processes: `clone`, `fork` and `vfork`. The `fork` system creates a full copy of the current process: the address space, page table, file descriptors and signal dispositions are copied (see [10] for further details). The current process becomes the new process's parent process. `vfork` behaves similar to `fork`, with the one exception that it does not copy the process's address space and page table. The `clone` system call is similar to `fork`, but allows fine-grained control over what parts of the process are duplicated.

Using the system call `execve` the execution of an executable is started. The address space and page table of the current process are emptied and the new executable is loaded. The kernel prepares the execution of a program by first filling the `linux_binprm` structure with the parameters used to execute the program, which includes the (executable) filename, the name of the interpreter (for binaries this equals the executable filename), the number of arguments and environment variables, credentials, a reference to the memory manager (used to map sections into memory). This structure is then passed on to the `search_binary_handler` function, which loops through all loaded `binfmt_*` kernel modules and asks them to load the binary. If none of the modules can successfully load the executable, the kernel returns `ENOEXEC`.

2.3.1 Overview of binary format handlers

Each implementation of a binary format handler (`binfmt` kernel module) can provide implementations to three different functions:

- `load_binary`: this function is used to load an executable binary, the `linux_binprm` structure is passed to this function. If the binary format handler cannot handle the given binary format, it simply returns `ENOEXEC`.

- load_shlib: this function is invoked by the system call handler for the uselib system call.
- core_dump: used to create memory dumps of processes. This function will not be discussed in detail in this work.

Note that both load_shlib and core_dump are not strictly required to be implemented by a binary format handler. An example would be the binfmt_misc, which can be used to implement executable format handlers by configuration: Users can add simple checks for magic values and provide a command that should be executed if the given magic value is found. Using this technique the proof of concept presented in this work can be configured to be called, if the user tries to execute an a.out binary.

In the case of a.out the inner workings of binfmt_aout can be summarised as follows:

1. Read the executable file header and determine the format.
2. Allocate enough pages for the text, data and bss sections.
3. Read the contents of the text and data sections.
4. Zero the bss segment.
5. Allocate and set up the stack.
6. Put the arguments (argc + argv) and environment variables (envc + envp) on the stack.
7. Set registers and start a new thread at the executable's entry-point (which usually equals the start of the text section).

For a.out libraries, which are loaded using the uselib system call, only steps 2 to 4 are necessary.

2.4 a.out Format Description

The a.out format in its simplest form consists of three parts: A 32 byte long header is followed by the executable instruction bytes (called the "text segment") and the initial static data bytes (called the "data segment").

After these three (mandatory) parts the binary may include (in that order) text relocation info, data relocation info and symbol and string tables used to store symbol/debug information. However, in this work we only focus on the mandatory sections supported on Linux and do not deal with BSD extensions.

2.4.1 The `a.out` Header

The structure of the `a.out` header is given with byte offsets in table 2.1[1, lines 5-15]:

bytes	0	1	2	3	4	5	6	7
0	magic		machine type	flags	.text segment length			
8	.data segment length				.bss segment length			
16	symbol table size				entry-point address			
24	text relocation table size				data relocation table size			

Table 2.1: `a.out` header fields including their byte offsets from the start of the file.

- **magic:** The "magic" byte sequence at the start of the executable file denotes the type of executable. For example, ELF binaries use the sequence "0x7fELF"[9]. In the case of `a.out` different two-byte sequences are used[2, lines 63-74]:
  - OMAGIC (`0407`)
  - NMAGIC (`0410`)
  - ZMAGIC (`0413`)
  - QMAGIC (`0314`)
  - CMAGIC (`0421`)

  Later, we will take a closer look at the meaning of these values. Note that we will not cover CMAGIC binaries, as these are only used for memory dumps and not executables.

  One interesting historical detail is that the magic number used for OMAGIC binaries `0407` matches the instruction encoding for a `br 7` instruction as understood by the PDP-11[5, page 4-37]. The Opcode is encoded in the first byte, the offset in the second byte. This is a primitive form of position-independent code, as a simple loader could load the program at address `0x0` and the first instruction would skip the header and jump right to the first instruction of the .text segment[16, page 51, footnote 1].

- **machine type:** Describes the format used for encoding the executable instruction bytes in the text segment. The following values are known to have been used. (see [2])
  - M_OLDSUN2 (`0x0`)

  - M_68010 (0x1)
  - M_68020 (0x2)
  - M_SPARC (0x3)
  - M_386 (0x64)
  - M_MIPS1 (0x97)
  - M_MIPS2 (0x98)

  In this work we will focus on the x86 processor family (M_386).

- **flags:** This field appears not to be used by the a.out format implementation provided by Linus Torvalds – it is expected to be zero [3, line 285] – however the BSD implementation supports two flags: EX_DYNAMIC and EX_PIC[11].
- **.text segment length:** Contains the length of the .text segment (given in bytes). This section's content starts directly after the 32 byte header.
- **.data segment length:** Contains the length of the .data segment (given in bytes). This section's content starts directly after the .text segment content.
- **.bss segment length:** Contains the length of the .bss segment (given in bytes). The acronym "bss" stands for "block started by symbol". The .bss is not present in the executable file itself. It is a zero-initialized extension of the .data section.
- **symbol table size:** Contains the size of the symbol table in bytes. a.out uses the STABS format.
- **entry-point address:** Contains the absolute entry-point address. Because Linux does not support relocations, this field is either 0 or contains a page-aligned address offset by 32 bytes. For a.out libraries this is the address at which the library should be loaded. For QMAGIC executables this is usually 0x1020.
- **text relocation table size:** Contains the size of the .text segment relocation table in bytes. Linux does not support relocations, therefore this field should be zero.
- **data relocation table size:** Contains the size of the .data segment relocation table in bytes. Linux does not support relocations, therefore this field should be zero.

2.4.2 The OMAGIC and NMAGIC format

OMAGIC is the simplest variant of a.out binaries. Binaries of this type are also called object files or "impure" executables; this hints at the fact that the sections are not aligned at page borders. The contents of the text and data sections are mapped into one contiguous segment at address 0.

The structure of NMAGIC binaries is similar to the OMAGIC format. At runtime the text section is also mapped at address 0, however, the data and bss sections are mapped at the start of the next page after the text section.

A big disadvantage of this is that the contents of the sections must be copied into memory. Directly mapping the sections into memory is not possible.

2.4.3 The ZMAGIC and QMAGIC format

In order to make it possible to directly map the sections into memory, the ZMAGIC format aligns all sections in the executable/object file at page boundaries.[16, page 53] That is, on a system with pages of 4KB all sections are expanded so that the size of each section is a multiple of 4KB. The size of the header is extended to 1024 bytes. Levine[16, page 53] mentions that the header is extended to 4KB as well, however, all ZMAGIC binaries that were analyzed while developing the run-aout prototype (as presented in this work) did not use a padding of 4KB in the header.

Of course, this wastes a lot of disk space: 992 bytes for the header and on average a 2KB gap between the text and data section. This led to the QMAGIC format, also called "compact pageable format"[16, page 53], which solves these issues by simply mapping the file header into the process. The entry-point is adjusted to include a 32 byte offset, skipping the header. For libraries the entry-point field also provides the base-address at which they should be loaded. For example, ld.so has the entry-point 0x62f00020 and therefore is mapped at 0x62f00000.

2.5 Analyzing the initialization sequence of an a.out binary

The first step was to analyze the successful execution of the a.out binary. Once we understood how it should behave, we could start emulating this behavior on a 64-bit system. It was possible to get to work a few select a.out QMAGIC binaries on a Ubuntu 16.04 LTS 32-bit with the legacy binfmt_aout kernel module. The memory mappings of these binaries/processes were then used as a baseline for the custom loader.

Looking at the output of strace in listing 2.2, we can see that first ld.so is loaded, then used to load other required libraries and then unloaded again. This behavior seems consistent across all the different kinds of a.out executables, some of which were disassembled in order to analyze them further.

Listing 2.2: (Shortened) output of strace showing the system calls that are executed

```
1 # strace ./usr/local/bin/gforth-0.3.0 -i slack/usr/local/lib/
     gforth/0.3.0/gforth.fi
2 execve("./usr/local/bin/gforth-0.3.0", ["./usr/local/bin/gforth
     -0.3.0", "-i", ''slack/usr/local/lib/gforth/0.3.0"...], [/*
     31 vars */]) = 0
```

```
3  uselib("/lib/ld.so")                    = -1 ENOENT (No such
       file or directory)
4  uselib("/usr/i486-linux/lib/ld.so")     = 0
5  stat("/etc/ld.so.cache", {st_mode=S_IFREG|0644, st_size=91963,
       ...}) = 0
6  open("/etc/ld.so.cache", O_RDONLY)      = 3
7  mmap(NULL, 91963, PROT_READ, MAP_SHARED, 3, 0) = 0xb7fe9000
8  close(3)                                = 0
9  write(2, "/lib/ld.so: cache '/etc/ld.so.ca"..., 55/lib/ld.so:
       cache '/etc/ld.so.cache' has wrong version
10 ) = 55
11 munmap(0xb7fe9000, 91963)               = 0
12 open("/etc/ld.so.conf", O_RDONLY)       = 3
13 fstat(3, {st_mode=S_IFREG|0644, st_size=72, ...}) = 0
14 read(3, "include /etc/ld.so.conf.d/*.conf"..., 72) = 72
15 close(3)                                = 0
16 uselib("include/libm.so.4")             = -1 ENOENT (No such
       file or directory)
17 uselib("/etc/ld.so.conf.d/*.conf/libm.so.4") = -1 ENOENT (No
       such file or directory)
18 uselib("include/libm.so.4")             = -1 ENOENT (No such
       file or directory)
19 uselib("/home/siegfried/usr/local/lib/libm.so.4") = 0
20 uselib("include/libc.so.4")             = -1 ENOENT (No such
       file or directory)
21 uselib("/etc/ld.so.conf.d/*.conf/libc.so.4") = -1 ENOENT (No
       such file or directory)
22 uselib("include/libc.so.4")             = -1 ENOENT (No such
       file or directory)
23 uselib("/home/siegfried/usr/local/lib/libc.so.4") = 0
24 munmap(0x62f00000, 20480)               = 0
25 ...
```

Listing 2.3: Output of pmap showing the memory mappings of an a.out binary, that loads additional libraries at runtime

```
1  # pmap 11898
2  11898:   ./usr/local/bin/gforth-0.3.0 -i slack/usr/local/lib/
       gforth/0.3.0/gforth.fi
3  00001000     28K r-x-- gforth-0.3.0
4  00008000      4K rwx-- gforth-0.3.0
5  00009000     96K rw---   [ anon ]
6  5ffff000    620K rwx-- libc.so.4.7.2
7  6009a000    188K rw---   [ anon ]
```

```
 8 600df000    108K rwx-- libm.so.4.6.27
 9 b7fbf000    260K rwx--   [ anon ]
10 b8001000     16K rwx--   [ anon ]
11 b8006000     16K rwx--   [ anon ]
12 b800b000     16K rwx--   [ anon ]
13 b8010000     16K rwx--   [ anon ]
14 bffde000    132K rw---   [ stack ]
15  total     1500K
```

In listing 2.3 we can see that `QMAGIC` binaries are loaded at address `0x1000` and all referenced libraries have hard-coded base addresses. The addresses used match the values in the `a_entry` fields in the respective `a.out` headers. We can also see that `ld.so` is unloaded after all other libraries are loaded.

As development was primarily done on a 64-bit system, the first step was to install support for running 32-bit executables and also adding the corresponding support for GCC. Then a small `a.out` binary that simply prints the message "Hello World!" was compiled using Netwide Assembler (NASM) as it is one of the few tools that still are able to produce `a.out` binaries natively.

In the first prototype the produced executable was loaded at address `0x10000` because it is the lowest possible address (see 3.3 for a more detailed explanation). This was done as a proof of concept, to test whether manually loading executable code is possible.

CHAPTER 3

Exploring different ideas: small steps towards a solution

This chapter discusses the different problems, ideas and approaches that were tried before arriving at the final solution discussed in the next chapter.

3.1 The Problem: Emulating the support for `a.out` on modern systems in usermode

In 2.3 we briefly discussed how a program is loaded by the kernel. The steps described there seem quite simple: Allocate process memory, load the sections of the executable into pages of memory and then kick off execution, by jumping to the entry-point of the executable. However, as it turns out, it is not quite that simple: The Linux kernel has gone through a lot of changes since the `a.out` format was replaced by `ELF` in 1995. A lot of new features and security measures were introduced, some of which make it harder if not impossible to run these binaries on modern systems:

1. The introduction of 64-bit and the subsequent removal of default support for 32-bit binaries on modern systems: Although this is easy to solve, it is something you have to think about, before being able to use 32-bit executables on Linux.
2. The implementation of Address Space Layout Randomization (ASLR) and restrictions regarding which parts of process memory may be used for executable purposes makes it impossible to load executable code at addresses smaller than `0x10000` (see 3.3 and 3.4).
3. The removal of the `binfmt_aout` kernel module necessitates the implementation of a custom handler for `uselib` system calls. This adds a lot of complexity.

3.2 Building an environment for successful execution of `a.out` binaries

The first problem we had to solve was finding a general architecture where `a.out` binaries could be loaded and executed successfully. Also, any incompatible or missing parts should be provided by this architecture.

3.2.1 Hosting a 32-bit executable in the lower address space of a 64-bit process

This idea was one of the first ideas that were explored: In the `x86_64` variant of the Linux kernel, all addresses with the most-significant bit (MSB) set to 0 are treated as user-space addresses; all addresses with the MSB set to 1 are treated as kernel-space addresses. So the idea was to use some address larger than `0xFFFF_FFFF` (but smaller than `0x8000_0000_0000_0000`) as the location of the loader code, that maps all relevant sections of the executable.

Of course this is only possible, if `x86` assembly and `AMD64` assembly are binary-compatible or it is possible to switch the CPU from 64-bit mode to 32-bit mode when jumping back and forth between the loader/trampoline code and the `a.out` executable code. However, `x86` assembly and `AMD64` assembly are not binary-compatible: Some `x86` opcodes were replaced by the so-called "REX prefixes" used in 64-bit mode to specify GPRs and SSE registers, 64-bit operand size and extended control registers[14, p. 2.2.1]. At least "[t]he single-byte-opcode form of [the] INC/DEC [is] not available in 64-bit mode." [14, 2.2.1.2, par. 2].

As experiments by others have shown[15], it is possible to mix 32-bit and 64-bit instructions in a 64-bit process in usermode, however the possibilities are very limited. Another problem is the incompatibility of `ld.so/uselib` system call handling in current kernels, as they no longer understand the `a.out` format, which made it necessary to intercept those system calls (see 3.5). For these reasons we decided to use a two-process model, where the `a.out` binary lives in a process separate from the loader.

3.2.2 Implementing a 32-bit loader using the C programming language

The idea was to implement all the binary parsing and loading logic using C and then using `fork` to create a copy of the whole process. After forking the `ptrace` API could be used to monitor the created process and handle any unsupported system calls.

However, as shown in listing 3.1, large parts of the address space would already be used by the host, which might prevent the `a.out` binary from properly executing. For example, the `run-aout` executable was placed in the range `0x565b_5000` to `0x565c_0000`. This address range, however, might be used by some `QMAGIC` executable library and because `a.out` libraries are not relocatable (at least not without making use of BSD-

specific extensions to the `a.out` format as described in 2.4), it is not possible to load libraries at other locations than specified in the `a_entry` field. Another problem is that the location of libraries required by the target executable might collide with the location of libraries used by the host.

In the final solution a handcrafted and carefully placed `trampoline` executable is used to avoid these problems. In particular the trampoline executable's code is moved to `0xc000_0000`, which is just above the 3GB mark.

Listing 3.1: Output of `pmap` showing the memory mappings of an early prototype of `run-aout`

```
1  # pmap 14524
2  14524:   ./run-aout ../exp/hello/test.o
3  0000000000010000      4K rwx-- test.o
4  00000000565b5000     12K r-x-- run-aout
5  00000000565b8000      4K r-x-- run-aout
6  00000000565b9000      4K rwx-- run-aout
7  00000000569ae000    136K rwx--   [ anon ]
8  00000000f7db4000   1864K r-x-- libc-2.27.so
9  00000000f7f86000      4K ----- libc-2.27.so
10 00000000f7f87000      8K r-x-- libc-2.27.so
11 00000000f7f89000      4K rwx-- libc-2.27.so
12 00000000f7f8a000     12K rwx--   [ anon ]
13 00000000f7fa8000      8K rwx--   [ anon ]
14 00000000f7faa000     12K r----   [ anon ]
15 00000000f7fad000      4K r-x--   [ anon ]
16 00000000f7fae000    152K r-x-- ld-2.27.so
17 00000000f7fd4000      4K r-x-- ld-2.27.so
18 00000000f7fd5000      4K rwx-- ld-2.27.so
19 00000000ffbb3000    132K rwx--   [ stack ]
20  total             2368K
```

3.3 Mapping memory at low addresses

Modern operating systems implement additional security measures protecting against memory access bugs and exploits. One such counter-measure is marking the first few pages of process memory as read-only. This, however, poses a problem for the execution of `a.out` binaries, as they expect the entry-point to be located at address 0 (or `0x1020` for `QMAGIC` binaries).

The solution to this problem was to override the kernel parameter `vm.mmap_min_addr`. The documentation on the kernel parameter `vm.mmap_min_addr`[19] states that "[t]his file indicates the amount of address space which a user process will be restricted from mmapping. [...] By default this value is set to 0 and no protections will be enforced by

the security module. Setting this value to something like 64k will allow the vast majority of applications to work correctly and provide defense in depth against future potential kernel bugs."

As a convenience feature run-aout will detect whether the `vm.mmap_min_addr` parameter is set correctly and warn the user about misconfiguration.

3.4 Address Space Layout Randomization

Another security measure protecting against process memory exploits is Address Space Layout Randomization (ASLR). It makes sure that future memory allocations can no longer be predicted or known from past executions of a program. However, some `a.out` programs may depend on absolute memory addresses, which is no longer possible with ASLR enabled, and would cause segmentation faults during execution. This problem can be solved by the use of Linux utilities like `setarch` or the `personality` API and setting the `ADDR_NO_RANDOMIZE` flag.

In later stages of development, we were not able to consistently reproduce the segmentation faults, due to the random factor. Also, turning off ASLR did not solve the problem consistently. Thus, we recommend executing `a.out` binaries using `root` privileges, as this would circumvent most security measures causing problems.

3.5 Binary patching

Binary patching is the process of modifying binaries to alter their behavior. The goal of this idea was to rewrite all occurrences of the `uselib` system call to instead jump to a fixed address in the same process to emulate it.

The assembly for the `uselib` system call used by `ld.so` can be seen in listing 3.2 and a possible patched version in listing 3.3. Thanks to the wide variety of instructions in `x86` and the use of a `mov r32, imm32` instruction in the original code, it is easily possible to inject a call to an emulation procedure.

Listing 3.2: Assembly for the invocation of the `uselib` system call as used by `ld.so`

```
1 00:     b8 56 00 00 00          mov    eax,0x56
2 05:     8b 5c 24 08             mov    ebx,DWORD PTR [esp+0x8]
3 09:     cd 80                   int    0x80
```

Listing 3.3: A possible patch replacing the system call with a call to our custom handler located at (for example) `0xc000_0000`

```
1 00:     b8 00 00 00 c0          mov    eax,0xc0000000
2 05:     8b 5c 24 08             mov    ebx,DWORD PTR [esp+0x8]
3 09:     ff d0                   call   eax
```

However, `ld.so` is not the only binary that needs this adjustment, in reality there are indefinitely many possible libraries that need this adjustment. Therefore it quickly became apparent that this approach can only serve as a stop-gap measure until a dynamic solution (using `ptrace`) is implemented.

CHAPTER 4

run-aout: A solution

Putting together all the pieces which we explored in the previous chapters, we are able to build a working example of a loader for `a.out` residing in user space. In the following chapter the architecture and implementation of `run-aout` will be discussed in detail.[1]

4.1 Overview

In general `run-aout` consists of two parts: the `run-aout` executable written in C and the `trampoline` executable written in assembly. The `run-aout` process (tracer/controller) loads the `a.out` binary file and checks whether all preconditions for successful execution are met. It then forks a child process (tracee/ trampoline) that is monitored using the `ptrace` API. The `trampoline` executable is loaded via `execve` and serves as host process for the `a.out` code and execution. Note that for simplicity, both the tracer and the trampoline are executed in 32-bit processes.

As previously discussed, `a.out` uses `ld.so` and the `uselib` system call to load libraries. As the Linux kernel no longer understands how to load `a.out` libraries, these system calls must be intercepted by the tracer and loading of `a.out` libraries must be emulated outside of the kernel.

[1] The source code of `run-aout` can be found at `https://github.com/siegfriedpammer/run-aout`

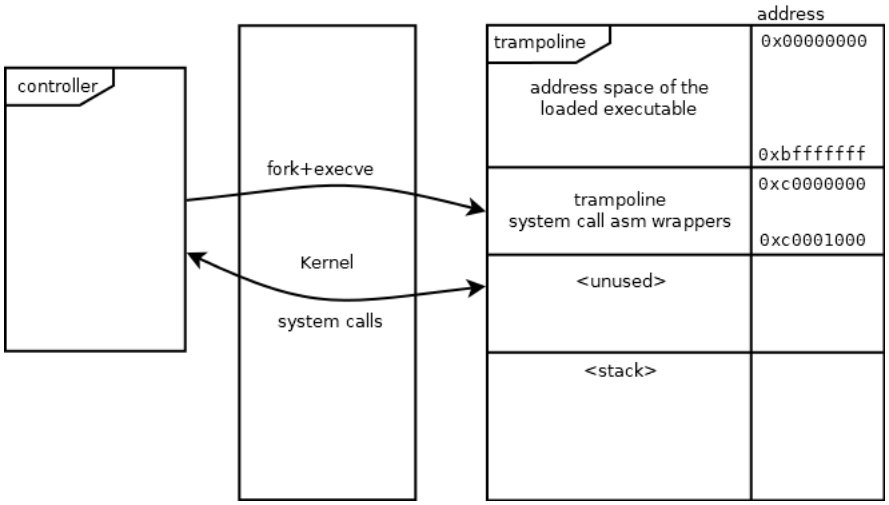

Figure 4.1: An overview of the processes involved in the emulation of `a.out`.

Before going into the implementation details, we will look at how the different components of `run-aout` solve the problems described in chapter 3:

1. The two-process model solves the problems described in 3.2.1 and 3.2.2 by separating the code that fulfills the role of the loader from the actual execution environment. Only a small portion of assembly code is used to mimic the actions the kernel would perform in-process.
2. The controller solves the memory mapping issues described in 3.3 by ensuring that it is either executed using `root` privileges, or that memory-mapping executable code at addresses below `0x10000` is possible, as described in 3.3.
3. It also enables the use of the `ptrace` API to handle `uselib` and other system calls, instead of having to patch the `uselib` system calls manually or semi-automatically, as described in 3.5.

4.2 The tracer/controller

The `run-aout` controller opens the given `a.out` binary and prepares it for execution. This preparation consists of the following steps:

First, it is ensured that either the process is executed as root or the kernel parameter `vm.mmap_min_addr` is set to a value smaller or equal to the page-aligned entry-point of the executable. That is `0x1000` for `QMAGIC` and `0x0` in all other cases. In case of `QMAGIC` the binary can be directly mapped into memory, however, in all other cases a temporary image file is written on-the-fly. In this temporary file, the individual sections are aligned to match the expectations of the executable format as described in 2.4.2 and 2.4.3:

- For ZMAGIC the first 1024 bytes are skipped and the start of the .text section is moved to offset 0. The .data section immediately follows the .text section, including a page-alignment padding.
- For OMAGIC the first 32 bytes (i.e., the executable header) are skipped and the .text section is moved to offset 0. The .data section immediately follows, without any padding.
- For NMAGIC the first 32 bytes (i.e., the executable header) are skipped and the .text section is moved to offset 0. The .data section is placed at the next address after that, which is a multiple of 1024 bytes.

The next step is to use fork to create a child process. The child process then uses the ptrace API PTRACE_TRACEME to allow the controller to trace it. Once tracing is started the child process uses execvp API to launch the trampoline. The controller detaches itself from stdin and sets the following options:

- PTRACE_O_TRACESYSGOOD: This is used to identify SIG_TRAPs caused by system calls.
- PTRACE_O_EXITKILL: This forces the child process to be killed as soon as the controller dies. This is necessary so that the child process may not escape the controller.

The controller waits for the execve system call to complete and then single-steps through the _start procedure. Once the call _syscall_mmap_exec instruction is reached execution is paused and the address, length and file descriptor are set to the EBX, ECX and EDX registers respectively. Then the child process is told to continue stepping.

After the process returns from the call, that is before the call _syscall_mmap_bss instruction, it is stopped again and if the a.out binary contains a .bss section greater zero, its address and length are written to the EBX and ECX registers, otherwise we increase the EIP register by 5 bytes and skip the instruction.

At this point the a.out binary should be mapped properly and its execution can be started. This is done by setting the absolute entry-point address to EAX and executing a jmp eax instruction. Now the initialization of the a.out binary execution is complete.

Now the controller waits for any uselib system calls and intercepts them. As described in section 7.3, there are two callbacks, for when a system call is entered and exited respectively. Once a uselib system call is detected, it is intercepted and the given library filename is loaded by the controller and checked for compatibility.

run-aout has a convenience feature, which makes it search a file named uselib.conf for library mappings. It may contain zero or more entries of the form: library_file.so :/path/to/library/file. This makes it possible to configure a.out libraries independently from the system's library search path.

4.3 The tracee/trampoline

The trampoline is a small set of wrapper functions around the system calls necessary to load `a.out` binaries:

- `_start`: This is the entry-point of the trampoline. Its main task is to open the `a.out` binary, map the relevant sections, prepare the stack and registers and then jump to the `a.out` entry-point.
- `_syscall_open`: A thin wrapper around the `open` system call. It expects the filename to be present in the `ebx` register. On success returns a file descriptor of the opened file.
- `_syscall_mmap`: A thin wrapper around the `mmap` system call.
- `_syscall_mmap_exec`: A thin wrapper around `_syscall_mmap` that maps the contents of the fd (given in `edx`) at an address and of some length (given in `ebx` and `ecx` respectively). This is used to map both `.text` and `.data` into one contiguous memory region.
- `_syscall_mmap_bss`: A thin wrapper around `_syscall_mmap` that creates an anonymous and "zeroed" mapping at an address and of length (given in `ebx` and `ecx` respectively).
- `_syscall_mmap_lib`: A thin wrapper around `_syscall_mmap` that emulates the `uselib` system call.

The inner workings of `_start` and `_syscall_mmap_lib` will be explained in detail in the following sections.

4.3.1 `_start`

`_start` is the entry-point of the `trampoline` binary. The controller launches the trampoline and hands over all command line arguments intended for the `a.out` binary. However, these binaries expect to find a "pointer table" on the stack[3, lines 54-59], as shown in table 4.1.

offset	content
esp+0	argc
esp+4	&&argv[0]
esp+8	&&envp[0]
...	...

Table 4.1: Arrangement of the stack as expected by `a.out`.

offset	content
esp+0	argc
esp+4	&argv[0]
esp+8	&argv[1]
...	...
esp+4*argc	&argv[argc - 1]
esp+4*(argc + 1)	NULL delimiter for argv
esp+4*(argc + 2)	&envp[0]
...	...

Table 4.2: Contents of the stack after the execution of the trampoline has started.

However, the stack/memory state produced by `execvp` provides the values of `argc`, `argv` and `envp` as a flat list, as shown in table 4.2. In order to achieve the result presented in table 4.1, we use the assembly code in listing 4.1.

Listing 4.1: Assembly used to prepare the pointer table required by `a.out` executables

```
1     sub esp, 12          ;; Make room for three 32-bit values
2     mov eax, [esp+12]    ;; argc is now stored at [esp+12], ...
3     mov [esp], eax       ;; ... copy it to [esp]
4     mov eax, esp         ;; Calculate the address of argv,
5     add eax, 16          ;; i.e., esp+16
6     mov [esp+4], eax     ;; and copy it to [esp+4], the 2. slot
7     mov eax, [esp]       ;; Calculate the address of envp,
8     add eax, 5           ;; i.e., (argc + 5) * 4 + esp, the
9                          ;; offset 5 is necessary, as
10    shl eax, 2           ;; there are the 3 pointers + argc +
11                         ;; the NULL terminator
12    add eax, esp         ;; of argv we need to skip.
13    mov [esp+8], eax     ;; and copy it to [esp+8],
14                         ;; the last slot
```

In line 1 space for 3 32-bit values (12 bytes) is allocated. Lines 2 and 3 simply copy the value of `argc` to `[esp+0]`. Lines 4 to 6 calculate the address of `argv[0]` and write it to `[esp+4]`. Lines 7 to 11 calculate the address of `envp[0]`.

As previously described, after this the next step is to map the `.text` and `.data`/`.bss` sections, and then jump to the entry-point of the `a.out` executable and start its execution.

4.3.2 _syscall_mmap_lib

The `uselib` system call has one argument (in `ebx`[21]): a pointer to the name of the library that should be loaded. After this system call is intercepted by the controller, and the file has been loaded and checked for compatibility, the controller invokes

_syscall_mmap_lib, which is located at address 0xc000_0000. The code in listing 4.2 shows the core of its implementation.

Listing 4.2: Assembly used to emulate the behavior of a uselib system call.

```
1     mov ebx, <full_library_filename>  ;; open file in ebx
2     call _syscall_open
3     cmp eax, 0                        ;; if the system call
4                                       ;; returns a number
5     jl _syscall_mmap_lib_exit_enoent  ;; smaller than 0,
6                                       ;; an error occurred.
7
8     mov edx, eax                      ;; execute the mmap
9                                       ;; system call to
10    mov ebx, <base_address>           ;; map the .text and
11                                      ;; .data sections.
12    mov ecx, <text_and_data_length>
13    call _syscall_mmap_exec
14    cmp eax, -4095                    ;; if -4095 is returned,
15    jae _syscall_mmap_lib_exit        ;; the system call failed.
16
17    mov ebx, <bss_address>
18    mov ecx, <bss_length>
19    cmp ecx, 0
20    je _syscall_mmap_lib_exit_success ;; check if the length of
21                                      ;; bss is > 0
22
23    call _syscall_mmap_bss            ;; allocate .bss
24    cmp eax, -4095
25    jae _syscall_mmap_lib_exit        ;; if an error occurred
26    jmp _syscall_mmap_lib_exit_success ;; exit, otherwise,
27 _syscall_mmap_lib_exit_enoent:         ;; return 0
28    mov eax, -2
29    jmp _syscall_mmap_lib_exit
30 _syscall_mmap_lib_exit_success:
31    mov eax, 0
```

One thing to note is that before this code is executed all values that match the <name> pattern are replaced by actual values as required to load the given library.

The meaning of these magic values is as follows:

- full_library_filename: the address of the filename of the library that should be loaded.

- `base_address`: the base address of the library.
- `text_and_data_length`: the combined size of `.text` and `.data` segments.
- `bss_address`: the address of the `.bss` segment, i.e., the library base address + the size of `.text` and `.data`.
- `bss_length`: the size of the `.bss` segment.

To sum it up, the code above first opens the library and then performs a mmap of `.text`, `.data` and (if necessary) `.bss` segments.

After successful execution, the controller receives the return value in `EAX` and before normal execution is continued it resets `eip` back to the return address of the system call and modifies the `EAX` register (system call result) according to the result of the emulation. Note that if the emulation is successful the "system call exit" is never performed – execution continues normally instead.

CHAPTER 5

Evaluation

In this section we will present the results of the execution of several `a.out` executables. The executables used are taken from a Slackware Distribution (dated circa 1994) and were extracted from two gzipped tarballs: `slack.tar.xz` and `usr.tar.xz`. Table 5.1 shows the names of executables, their location inside the package, the type, file size and whether the execution of the executable using `run-aout` was successful.

name	package	path	type	size	success
pwd	slack.tar.xz	bin/pwd	OMAGIC	1368	yes
echo	slack.tar.xz	bin/echo	OMAGIC	3312	yes
ls	slack.tar.xz	bin/ls	ZMAGIC	25604	yes
mkdir	slack.tar.xz	bin/mkdir	ZMAGIC	13316	yes
ping	slack.tar.xz	bin/ping	OMAGIC	10860	yes
gforth-0.2.1	usr.tar.xz	local/bin/gforth-0.2.1	QMAGIC	32768	no
gforth-0.3.0	usr.tar.xz	local/bin/gforth-0.3.0	QMAGIC	32768	yes
Mosaic	usr.tar.xz	local/bin/Mosaic	ZMAGIC	1551772	no
dot	usr.tar.xz	local/bin/dot	ZMAGIC	264675	no

Table 5.1: Overview of all tested and analyzed executables

The libraries used for testing were mapped as follows:

Listing 5.1: Contents of uselib.conf

```
1 ld.so:/home/siegfried/bachelorarbeit/lib/ld.so
2 libc.so.4:/home/siegfried/bachelorarbeit/lib/libc.so.4.7.2
3 libm.so.4:/home/siegfried/bachelorarbeit/lib/libm.so.4.6.27
```

5.1 Analyzing the failed executions

`run-aout` was developed with extensive logging support, which allows deep analysis of the failures during execution of an `a.out` executable. The types of errors fall into two categories:

1. Problems with system calls, that need special support: the `brk` system call is sometimes used to allocate memory by changing the size of the data segment. This causes problems because the program break is located after the end of the trampoline (at `0xc000_1000`) and `brk` will return an address after that.
2. The use of libraries using formats other than QMAGIC is simply not yet implemented in `run-aout`. The Mosaic executable is such an example. It uses `libXt.so.6.0`, which is a library stored in the ZMAGIC format. However, loading would require the `run-aout uselib` handler to convert the ZMAGIC library on the fly (just like it is done with ZMAGIC executables) but then store them on disk, so the uselib handler of the trampoline is able to load the binary. For simplicity we only implemented QMAGIC libraries.

5.1.1 gforth-0.2.1

Listing 5.2: Output produced by `gforth 0.2.1`

```
1 ./run-aout -- ../gforth/gforth-0.2.1 -i ../gforth/gforth-0.2.1.
     fi
2 /lib/ld.so: cache '/etc/ld.so.cache' has wrong version
3 ../gforth/gforth-0.2.1: Cannot load nonrelocatable image (
     compiled for address $11030) at address $c0960030
4 The Gforth installer should look into the INSTALL file
```

The execution of `gforth-0.2.1` fails because it tries to allocate space for the `gforth-0.2.1.fi` image using `brk`, which returns an address above `0xc000_0000`. This can be seen in the output of `pmap` just before the `gforth-0.2.1` program terminates:

Listing 5.3: Pmap for `gforth 0.2.1`

```
1  18488:   ../gforth/gforth-0.2.1 -i ../gforth/gforth-0.2.1.fi
2  0000000000001000     32K rwx-- gforth-0.2.1
3  000000005ffff000    620K rwx-- libc.so.4.7.2
4  000000006009a000    188K rwx--   [ anon ]
5  00000000600df000    108K rwx-- libm.so.4.6.27
6  00000000c0000000      4K r-x-- trampoline
7  00000000c0f9b000    368K rwx--   [ anon ]
8  00000000f7f3d000     12K r----   [ anon ]
9  00000000f7f40000      4K r-x--   [ anon ]
10 00000000ff7e0000    132K rwx--   [ stack ]
```

```
11  total                1468K
```

The segment starting at `0xc0f9_b000` is the segment that `gforth-0.2.1` tries to load the image at. This fails, however, as the image is not relocatable and the expected address is `0x0001_1030`.

5.1.2 Mosaic

Mosaic is the most complex program we tried to execute. The output of the first execution is as follows:

Listing 5.4: Output of the first execution of Mosaic

```
1 # ./run-aout -- ../../ertl/usr/local/bin/Mosaic
2 /lib/ld.so: cache '/etc/ld.so.cache' has wrong version
3 ../../ertl/usr/local/bin/Mosaic: can't find library 'libXt.so.6'
```

After adding `libXt.so.6` to `uselib.conf` mapped to `/home/siegfried/ertl/usr/local/lib/libXt.so.6.0`. The program continues to execute, but `libXt.so.6.0` is a library in the ZMAGIC format, which is not yet implemented and thus its execution fails. Adding support for ZMAGIC binaries seems feasible, but might involve a bit more work.

CHAPTER 6

Related Work

6.1 Emulation

Emulation is used to provide a suitable environment for the execution of legacy software and hardware systems by imitating the behavior of legacy systems. [8] A. Doruk et al. give an overview of the different approaches to emulation. They mention the distinction between a "System Virtual Machine" (examples include VirtualBox, VMWare, HyperV) and a "Process Virtual Machine" (examples would be QEMU, Wine). The `run-aout` prototype would be best classified as a "Process Virtual Machine", because it "is developed for handling only one process. It is used for running the process that is compatible with different environment on current system. It is created, when the process starts and killed, when the process finishes." [8, p. 2]

In his work "An X86 emulator written in Java" [4] J. Burcham focuses on the emulation of a complete x86 environment, including process creation, binary loading and instruction decoding and translation. Most of these topics were not relevant, as we tried to provide a simple and fast mechanism to execute programs stored in a deprecated format. The format served as a container for a compatible x86 instruction and data stream. We were able to leverage the (still) existing support for the x86 architecture built into Linux and only had to fill the gaps to allow the execution of instructions. A similarity both approaches share is the need to understand the structure of the binary format.

6.2 Executable formats

A very good and complete reference about all sorts of binary executable formats is John R. Levine's "Linkers & Loaders" [16]. Using this and the manual page for `a.out.h`[11] provided us with all the information needed to get started. As explained in 2.4.3 some details described by Levine did not match our observations, nonetheless, his work proved invaluable in the realization of the `run-aout` prototype.

In their work “SRL - A Simple Retargetable Loader” [20], Ung and Cifuentes also deal with binary and executable formats, but from a different angle: the goal of their work is to eliminate the need to provide a loader for each format on each platform. Instead of manually creating a loader for each platform/format combination, they want to automate the procedure and generate the loader from a format specification, for which they define a grammar.

CHAPTER 7

Conclusion and Future Work

Although this work provides a proof that loading and executing `a.out` binaries in user mode Linux is possible, there are some issues, which still need to be addressed in the future:

- We observed problems with certain system calls, which lead to crashes and segmentation faults during execution. For example, programs that use the `brk` system call to allocate memory by changing the size of the data segment, might run into problems due to how the process created by `run-aout` works.
  The code used by the trampoline is located at `0xc000_0000` and `brk` will return an address after the end of the trampoline's `.text` segment. However, `a.out` programs might expect it to return an address after the `a.out` program's `.data` section, which is no longer possible.
  Solving this problem will require `run-aout` to emulate the original behavior of the `brk` system call.

- If Linux ever decides to completely remove native support for 32-bit processes and binaries, just as it happened to the built-in `a.out` support, additional work may be required to make `run-aout` work with existing emulation software such as QEMU.

- Another problem is that the current implementation of `run-aout` makes debugging the `a.out` program very difficult. A process can only be traced by exactly one process. However, GDB offers the "GDB remote protocol" [12] which could be used to exchange commands and data between GDB and the `a.out` process.

Bibliography

[1] *a.out.h source code part 1.* URL: https://github.com/torvalds/linux/blob/80145ac2/arch/x86/include/uapi/asm/a.out.h#L5-L15 (visited on Nov. 24, 2020).

[2] *a.out.h source code part 2.* URL: https://github.com/torvalds/linux/blob/80145ac2/include/uapi/linux/a.out.h (visited on Nov. 24, 2020).

[3] *binfmt_aout source code.* URL: https://github.com/torvalds/linux/blob/80145ac2/fs/binfmt_aout.c (visited on Nov. 24, 2020).

[4] Jonathan Kenneth William Burcham. "An X86 emulator written using Java". MA thesis. University of Manchester, Faculty of Engineering and Physical Sciences, School of Computer Science, 2005.

[5] Digital Equipment Corporation. *PDP-11/40 processor handbook.* https://pdos.csail.mit.edu/6.828/2005/readings/pdp11-40.pdf. Digital Equipment Corporation, 1972.

[6] Digital Equipment Corporation. *PDP-11/45 processor handbook.* https://onedrive.live.com/?authkey=%21AOYnMh16d4YDKq4&cid=7D567B5161FE0EB7&id=7D567B5161FE0EB7%2136652&parId=7D567B5161FE0EB7%2136608&o=OneUp. Digital Equipment Corporation, 1973.

[7] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel, 3rd Edition.* O'Reilly, 2005.

[8] Alpay Doruk and H Nusret Buluş. "The Design of an Emulator". In: *Proceedings of UNITECH 2017, Gabrovo, II-265.* International Scientific Conference (UNITECH 2017, Gabrovo), 2017.

[9] *elf man page.* URL: https://man7.org/linux/man-pages/man5/elf.5.html (visited on Nov. 24, 2020).

[10] *fork man page.* URL: https://man7.org/linux/man-pages/man2/fork.2.html (visited on Jan. 20, 2021).

[11] *FreeBSD a.out man page.* URL: https://www.freebsd.org/cgi/man.cgi?a.out(5) (visited on Nov. 24, 2020).

[12] *GDB Remote Protocol documentation*. URL: https://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Protocol.html (visited on Jan. 22, 2021).

[13] Dennis M. Ritchie. "The Development of the C Language". In: *History of Programming Languages (HOPL-II) Preprints*. SIGPLAN Notices 28(3). ACM Press/Addison-Wesley, 1993.

[14] *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 2A: Instruction Set Reference, A-M*. Accessed: 2020-11-11. URL: https://www.intel.com/content/dam/support/us/en/documents/processors/pentium4/sb/25366621.pdf.

[15] *Is it possible to use both 64 bit and 32 bit instructions in the same executable in 64 bit Linux?* URL: https://stackoverflow.com/a/48855022/3357229 (visited on Dec. 29, 2020).

[16] John R. Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, San Francisco, California, 2000.

[17] *Linux Kernel Finally Deprecating A.out Support*. URL: https://www.phoronix.com/scan.php?page=news_item&px=Linux-Dropping-A.Out (visited on Dec. 27, 2020).

[18] *Linux Kernel git repository*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=eac616557050737a8d6ef6fe0322d0980ff0ffde (visited on Dec. 27, 2020).

[19] *mmap_min_addr | sysctl-explorer.net*. URL: https://sysctl-explorer.net/vm/mmap_min_addr/ (visited on Nov. 10, 2020).

[20] David Ung and Cristina Cifuentes. "SRL 3/4-A Simple Retargetable Loader". In: *1997 Australian Software Engineering Conference (ASWEC '97), 28 September - 2 October 1997, Sydney, Australia*. IEEE Computer Society, 1997, pp. 60–69. DOI: 10.1109/ASWEC.1997.623755. URL: https://doi.org/10.1109/ASWEC.1997.623755.

[21] *x86 syscall reference sheet*. URL: https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit (visited on Nov. 24, 2020).

[22] *x86: Deprecate a.out support*. URL: https://patchwork.kernel.org/project/linux-fsdevel/patch/20190305145717.GD8256@zn.tnic/ (visited on Mar. 8, 2021).

List of Figures

List of Tables

Listings

Appendix

7.1 Implementation of `run-aout`

For the source code of the implementation and all related material see the Git repository hosted at `https://github.com/siegfriedpammer/run-aout`.

7.2 x86 Architecture

The x86 architecture is a CISC (Complex Instruction Set Computer) architecture, as it offers a wide variety of instructions, which are 1 up to 15 bytes in length. Modern x86 processors support different modes of operation, including "real mode" (16-bit), "protected mode" (32-bit) and "long mode" (64-bit). Long mode supports both 64-bit and 32-bit programs.

In 32-bit mode there are nine registers:

- `EAX`: Accumulator, usually contains procedure return values
- `EBX`: Base index
- `ECX`: Counter
- `EDX`: Extend accumulator
- `ESI`: Source index
- `EDI`: Destination index
- `ESP`: Stack pointer (top address)
- `EBP`: Stack base pointer (holds current stack frame address)
- `EIP`: Instruction pointer

All of these can be used freely (except `EIP`), which can only be modified by branch instructions. `ESP` and `EBP` are typically modified at the start/end of a procedure to allocate/free space for additional values.

Additionally, there are six segment registers `CS` (code segment), `DS` (data segment), `SS` (stack segment), `ES` (extra segment), `FS` (extra segment 2), `GS` (extra segment 3), which were introduced in 16-bit real mode, to allow the use of more than 64KB of memory (similar to PDP-11's I space and D space, albeit more advanced). In the 32-bit and 64-bit architecture the segment registers are no longer used, but are kept for backward compatibility.

The special `EFLAGS` register provides information on the state of execution, such as the `ZF` (zero flag) or `OF` (overflow flag) and many others.

7.3 System Calls

System calls are used in many operating systems as a way for user-space code to call into the kernel. Many functions in the C standard library are wrappers around system calls, such as `open`, `read`, `write` or `close`.

In Linux there are multiple ways of executing system calls[7, Chapter 10]: "legacy" and "fast" system calls. The fast system calls use dedicated instructions: `sysenter` and `sysexit` for 32-bit, `system call` and `sysret` for 64-bit. We will focus on "legacy" system calls, because they were used in the binaries that were used as examples.

Legacy system calls are performed using the `int 0x80` instruction. Each system call has a unique ID, which is stored in the `EAX` register. Up to six arguments can be directly stored in the `EBX`, `ECX`, `EDX`, `ESI`, `EDI` and `EBP` registers. Some system calls (e.g., `mmap`) require that the arguments must be passed on the stack, with the stack address passed in the `EBX` register. The result of the system call is stored in the `EAX` register before control is returned to user space.

After the interrupt is received, the current register state is saved on the kernel's stack and then the kernel uses the system call number stored in `EAX` to jump to the appropriate handler. After the call is completed the original register state (with one exception: `EAX` contains the result of the system call) is restored and execution of the user code process continues.