UNIVERSITY OF GENOVA, DIBRIS

Binary Analysis and Secure Coding

Andrea Basso

May 2, 2019

Contents

Chapter 1

Binary Analysis

Binary analysis can be carried out with two different techniques:

- Static Analysis:
  - The entire binary can be potentially analysed in one go;
  - No need for an architecture capable of running the binary;
  - No knowledge of the runtime states;
- Dynamic Analysis:
  - Often simpler;
  - Possibility to observe a particular execution;
  - Chance to miss interesting parts of the code;
  - Potentially dangerous if analysisng malware;

It can present several challenges as:

- it is in a low level language - i.e. Assembly - that can present itself in different flavours (AT&T, Intel, non x86 platforms);
- There might be no symbols (e.g. stripped binaries);
- Code and data are mixed together;
- Impossible to remove/add instructions without "bricking" the program.

1.1 Execution

Unless one is programming the bare metal, programs are generally executed on an Operating System (OS hereafter) which is an **abstract machine**. A running profram is a process, which has its own resources and can use a subset of the ISA (implemented on the actual CPU or emulates), an abstract model of a CPU and the API of the OS (e.g. syscall).

1.1.1 Emulation, Virtualisation, Containers

As computers became more accessible to people the idea of simply time sharing the machine became unfeasible; system had to be made reduntant and fault tolerant whilst being able to be shared across the entire userbase. Having multiple systems is beneficial for:

- **Isolation** Systems must be isolated for stability reasons in such a way that programs do not interfere with each other for stability (i.e. buggy programs shall not cause disruption to other processes);
- **Performance** Placing an application on it's own system allows it to have exclusive access to the system's resources. User level segregation does not effectively isolate applications as scheduling priority, memory demand, network and FS I/O can generate interference between programs.

However having multiple systems is not always the most fruitful and effective solution and in the 60's IBM started working on the first *virtual machines* where it was crucial to be able to time-share expensive main frames. A VM is a fully isolated copy of the underlying hardware that enable applications to run as they were on "bare metal".

Virtual Machine Monitor

The VMM is the software component that hosts the virtual machines; the VMM is generally called *host* and the VM's are called *guests*. It's duties are, among the others, the provision of virtulised CPU, virtualised resources such as I/O devices, storage, memory and isolation between VM's.

Internet Services Virtualisation

In order to face the ever more demanding Internet applications a new kind of system was adopted: the multi-tier system architecture. THis common way of building Internet applications entails the separation of the web and database server and easily enables load-balancing and clustering. The system has the advantage of being easier to manage and extremely fault tolerant, but this requires the additional cost of additional hardware purchases and management. In this scenario the system virtualisation provides the advantages of componitisation without the need for additional hardware (and exacerbated by Zipf's law[1]). Studies have found that the internet services behave according to a Zipfian distribution, i.e. if we observe the web caches we see that only a few web services are always active and used the most of the others just lay dormant; thus isolation is no longer a worthy option unless services are consolidated to a single machine and virtualisation provides the best benefit for the cost as it avoids the waste of computing power and reduces the maintenance costs.

Requirements for VM's

In 1974 Popek and Goldberg defined what they believed where the formal requirements for a virtualisable computer architecture: *"For any computer a virtual machine monitor can be constructed if the set of instructions is a subset of the privileged instructions"*; the most essential requirement a computer architecture must have in order to be virtualisable is that privileged instructions must launch a trap and render control to the VMM so it can decide whether the instruction can be executed or not. The three essential characteristics of VM's are:

1. Any program run under the VMM should exhibit an effect identical with that demonstrated if the program had run on the original machine (with the exception of the timing effects related to the virtualisation overhead);
2. A statistically dominant subset of the virtual processor's instructions are directly executed by the real processor - i.e. the virtual machine occasionally relinquishes the real processor to the virtual processor;
3. The VMM is in complete control of system resources and the virtual machine does not have direct access to the system's real resources.

[1] The frequency of an event is proportional to $x^{-\alpha}$ where x is the rank of the event in comparison to other events

1.1.2 Full System Virtualisation

Full sytem virtualisation provides a virtual replica of the system's hardware so that the OS's and software may run on the VM exactly as it would on "bare metal".

370-XA Virtualisation The VM/370 is a VMM that was first delivered in the VMF/370 and ran the 370-XA, an architecture designed to maximise the virtualisation performance. The platform provided *"assists"* in the architecture to boost the virtualisation performance of certain repetitive operations. The VMM had to simulate the instruction beforehand to ensure safety; the assist development allowed the execution of the simulated instructions on the hardware. Other assists supplanted the frequently executed instructions that still required VMM intervention and were enabled by setting them in *interpretive execution* mode. Assists included VMA, ECPS, STB and many others (IBM developed more than 100 in total).

IA-32 It was the most dominating architecture (now supplanted by x64) but it was never meant to be virtualised although that would have had enormous benefits; the reason behind this resides in the fact that there is a limited subset of instructions that does not need higher privilege to be run and cripple a machine. This is not a problem in systems with single OS, but it becomes difficult to manage when multiple OS's are running at once. Furthermore the huge amount of different devices and drivers for the IA-32 made virtualisation a big deal. However IA-32 has been finally virtualised according to the following

1. Non-sensitive, non-privileged instructions may be run directly on the processor;
2. Sensitive, Privileged instructions trap - i.e. the processor sends an interrupt and the VMM instercepts it for proper management of the instruction
3. Sensitive, but not privileged instructions are detected and the VMM ensures they never get executed as they cannot be handled with a trap.

IA-64 It presents more or less the same drawbacks of the IA-32, but it provides an important feature as it allows to run VM's in a higher ring of the VMM and traps from the higher rings can be intercepted by lower rings.

Drawbacks of full system virtualisation

The advantage of full system virtualisation is that OS's and programs can run unmodified and completely oblivious of the environment in which they are actually running. However there are drawbacks related to the fact that IA-32 and IA-64 were not created to be virtualised and thus a lot of effort must be put into efficiently virtualising hardware and memory management. When an application demands for a memory page the system translates the virtual address to a physiscal one thanks to page table. Unused pages can be written to disk when inactive or when more memory is required. The VMM must intercept all of the memory calls and translate VM space into the system's real space using another pag table, retrieve the memory and return it to the VM.

1.1.3 Paravirtualisation

Paravirtualisation tries to mitigate the performance hit by modifying the OS so that tasks are performed by the VMM directly and not by the CPU. There are several commecrcial and open solutions that can efficiently carry out paravirtualisation like:

- Xen Project
- Citrix Xen Server
- Denali

- VMWare Sphere

The VMM provides the VM's with a software interface that is similar yet not entirely identical to the underlying hardware. Paravirtualisation provides specifically defined hooks by which it tries to minimise the number of operations that must be carried out in the virtual environment. The guest OS shall be modified for the paravirtualisation API, although this requirement is less strict nowadays as more and more paravirtualsation enabled components can be found under the GPL and other free licenses.

1.1.4 Other virtualisation techniques

There are other virtualisation techniques like:

- Emulation: replication of a specific architecture that enables a machine to behave like another
- Containerisation: OS-level virtualisation that allows for the presence of isolated user-space instances called containers.

1.2 Binaries

Executables are created in compliance with a specific ABI that depends on the architecture. The ABI establishes:

- The executable and object file formats
- fundamental types (size alignment for int, long and other primitives)
- How data types are laid out in memory
- Function and system call conventions
- How programs are started up (initialisation, dynamic linking, etc.)

1.2.1 Executable and Linkable Format

It is a flexible format that can be used for:

- executables;
- dynamic libraries (*.so);
- object files (also called relocatable files).

The ELF format is a sequence of headers, segments and sections containing the necessary information and data for the execution and linkage of the executable.

File Header

The file header is located at the beginning of the file and is used to locate the other parts.

```
typedef struct {
  unsigned char e_ident;
  Elf64_Half    e_type;
  Elf64_Half    e_machine;
  Elf64_Word    e_version;
  Elf64_Addr    e_entry;
  Elf64_Off     e_phoff;
  Elf64_Off     e_shoff;
  Elf64_Word    e_flags;
  Elf64_Half    e_ehsize;
  Elf64_Half    e_phentsize;
  Elf64_Half    e_phnum;
  Elf64_Half    e_shentsize;
  Elf64_Half    e_shnum;
  Elf64_Half    e_shstrndx;
} Elf64_Ehdr;
```

Listing 1.1: File Header

where:

- `e_ident` is an array of bytes that identifies the file as an ELF and provides info about the data representation of the object file structures. The bytes of the identifier are:

  `e_ident[EI_MAG0]`, `e_ident[EI_MAG1]`, `e_ident[EI_MAG2]`, `e_ident[EI_MAG3]` contain the so called magic number '0x7fELF';

  `e_ident[EI_CLASS]` identifies the class of the ELF (32bit or 64bit);

  `e_ident[EI_DATA]` specifies the encoding of the object file data structures (little or big endian);

  `e_ident[EI_VERSION]` identifies the version of the object file format and is presently set to EV_CURRENT which has value 1;

  `e_ident[EI_OSABI]` identifies the OS and ABI version for which the object file is prepared. Some fields in other ELF structures have flags that can be interpreted based on the value of this field

  `e_ident[EI_ABIVERSION]` Identifies te version of the ABI fdor which the binary is prepared. The interpretation of this field depends on the value of the `eident[EI_OSABI]`

- `e_type` Identifies the object file type and its values can be:

Name	Value	Purpose
ET_NONE	0x0000	No file type
ET_REL	0x0001	Relocatable file type
ET_EXEC	0x0002	Executabler
ET_DYN	0x0003	Shared Object
ET_CORE	0x0004	Core File
ET_LOOS	0xFE00	Environment specific use
ET_HIOS	0xFEFF	Environment specific use
ET_LOPROC	0xFF00	Processor specific use
ET_HIPROC	0xFFFF	Processor specific use

- `e_machine` identifies the architecture
- `e_version` identifies the version of the current file format
- `e_entry` contains the virtual address of the entry point

- `e_phoff` contains the file offset in bytes of the program header table
- `e_shoff` contains the offset in bytes of the section header table
- `e_flags` contains processor specific flags
- `e_ehsize` contains the size in bytes of the ELF header
- `e_phentsize` contains the size in bytes of a progam header entry
- `e_phnum` contains the number pf entries in the program header table
- `e_shentsize` contains the size in bytes of a section headeer table entry
- `e_shnum` contains the number of entries in the section header table
- `e_shstrndx` section header table index of the string table

Sections

The code and data in an ELF binary are logically divided into contiguous nonoverlapping chunks called sections; their structure is not predetrmined and varies depending on the content. Sections contain all the information in an ELF file, with the exception of the file header and can be identified by their index into the section header table. Strictly speaking the organisation into sections is intended to be convenient at linking; hence not all of the sections are actually needed while setting up a process and the virtual memory (e.g. symbols and relocation sections do not contain info needed during execution). As sections are only there for the linker the presence of the section header table is not mandatory (and the `e_shoff` is set to 0 in this case).

Section Indices Section indices 0, 0xFF00-0xFFFF are reserved for special purposes as per the following:

Name	Value	Description
SHN_UNDEF	0	Used to mark undefined or meaningless section reference
SHN_LOPROC	0xFF00	Processor-specific use
SHN_HIPROC	0XFF1F	Processor-specific use
SHN_LOOS	0xFF20	Environment-specific use
SHN_HIOS	0xFF3F	Environment-specific use
SHN_ABS	0xFFF1	Indicates that the reference is an absolute value
SHN_COMMON	0xFFF2	Indicates a symbol that has been declared as a common block (tentative declaration in C)

The first entry in the section header table (at index 0) **MUST** contain all zeroes.

Section Header Entries The structure of a section header is as per the following

```
typedef  struct {
  Elf64_Word sh_name;
  Elf64_Word sh_type;
  Elf64_Xword sh_flags;
  Elf64_Addr sh_addr;
  Elf64_Off sh_offset;
  Elf64_Xword sh_size;
  Elf64_Word sh_link;
  Elf64_Word sh_info;
  Elf64_Xword sh_addralign;
```

```
    Elf64_Xword  sh_entsize;
12 } Elf64_Shdr;
```

Listing 1.2: Section Header

where:

- `sh_name` contains the offset, in bytes, to the section name relatively to the start of the section name string table;
- `sh_type` identifies the section type detailed as follows

Name	Value	Meaning
SHT_NULL	0	Marks an unused section header
SHT_PROGBITS	1	Contains program data such as machine instructions and constants
SHT_SYMTAB	2	Contains the static linking symbol table
SHT_STRTAB	3	Contains a string table
SHT_RELA	4	Contains "Rela" type relocation entries
SHT_HASH	5	Contains a symbol hash table
SHT_DYNAMIC	6	Contains dynamic linking tables
SHT_NOTE	7	Contains note information
SHT_NOBITS	8	Contains uninitialised space; does not occupy any space in the file
SHT_REL	9	Contains "Rel" type relocation entries
SHT_SHLIB	10	Reserved
SHT_DYNSYM	11	Contains a dynamic loader symbol table
SHT_LOOS	0x6000 0000	Environment-specific use
SHT_HIOS	0x6FFF FFFF	Environment-specific use
SHT_LOPROC	0x7000 0000	Processor-specific use
SHT_HIPROC	0x7FFF FFFF	Processor-specific use

- `sh_flags` identifies the attributes of the section name as listed:

Name	Value	Meaning
SHF_WRITE	0x1	Contains writeable data
SHF_ALLOC	0x2	Section allocated in memory image of a program
SHF_EXECINSTR	0x4	Contains executable instructions
SHF_MASKOS	0x0F00 0000	Environment-specific use
SHF_MASKPROC	0xF000 0000	Processor-specific use

- `sh_addr` contains the virtual address of the beginning of the section in memory. If the section is not allocated in memory this shall be 0;
- `sh_offset` contains the offset of the section in bytes from the beginning of the file;
- `sh_size` contains the size of the section (unless it's a `SHT_NOBITS`);
- `sh_link` contains the index of an associated section. This field is used for several purposes as detailed in the following:

Type	Associated Section
SHT_DYNAMIC	String table used by entries in this section
SHT_HASH	Symbol table in which the hash table applies
SHT_REL/RELA	Symbol table referenced by the relocations
SHT_SYMTAB/DYNSYM	String table used in entries used in these sections

- `sh_info` contains extra info about the section as explained below:

Type	Info
SHT_REL/RELA	Index of the section which the relocation applies to
SHT_SYMTAB/DYNSYM	Index of the first non-local symbol (i.e. number of local symbols)

- `sh_addralign` contains the required alignment of the section and must be a power of 2;
- `sh_entsize` contains the size in bytes, of each entry, for sections that contain fixed-size entries (zero otherwise).

The standard sections for Code/Data and other object file are reported in tables 1.2.1 and 1.2.2.

Name	Type	Flags	Use
.bss	SHT_NOBITS	SHF_ALLOC/WRITE	Uninitialised data
.data	SHT_PROGBITS	SHF_ALLOC/WRITE	Initialised data
.interp	SHT_PROGBITS	SHF_ALLOC	Program interpreter path name
.rodata	SHT_PROGBITS	SHF_ALLOC	Read only data (const and literals)
.text	SHT_PROGBITS	SHF_ALLOC/EXECINSTR	Executable code

Table 1.2.1: Code and Data Standard Sections

Name	Type	Flags	Use
.comment	SHT_PROGBITS	None	Version Control Info
.dynamic	SHT_DYNAMIC	SHF_ALLOC/WRITE	Dynamic Linking Tables
.dynstr	SHT_STRTABS	SHF_ALLOC	String table for the `.dynamic` segment
.dynsym	SHT_DYNSYM	SHF_ALLOC	Read only data (const and literals)
.got	SHT_PROGBITS	machine dependent	Global Offset Table
.hash	SHT_HASH	SHT_ALLOC	Symbol Hash Table
.note	SHT_NOTE	none	Note Section
.plt	SHT_PROGBITS	machine dependent	Procedure Linkage Table
.rel[*name*]	SHT_REL	SHT_ALLOC	Relocations for section [*name*]
.rela[*name*]	SHT_RELA	SHT_ALLOC	As Above
.shstrtab	SHT_STRTAB	None	Section Name String Table
.strtab	SHT_STRTAB	None	String Table
.symtab	SHT_SYMTAB	SHT_ALLOC	Linker Symbol Table

Table 1.2.2: Other Object Files Standard Sections

1.2.2 String Table

Contains the strings used for section and symbol names. It is an array of null terminated strings. Section header table entries, and symbol table entries refer to strings in a string table with an index relative to the beginning of the string table. The first byte of the table must be null so that the index 0 always refers to a non-existent name.

1.2.3 Symbol Table

The first symbol table entry is reserved and must be all zeroes (`STN_UNDEF` is useed as a reference to this entry). The structure of a symbol table entry is

```
typedef  struct {
  Elf64_Word sh_name;
  unsigned char   st_info;
  unsigned char   st_other;
  Elf64_Half    st_shndx;
  Elf64_Addr    st_value;
  Elf64_Xword   st_size;
} Elf64_Sym;
```

Listing 1.3: Symbol Table Entry

where:

- `st_name` contains the offset in bytes to the symbol name from the beginning of the string table;
- `st_info` contains the symbol type and its binding attributes - i.e. the scope. The binding attributes are contained in the high-order 4 bits of the eight bit byte and the symbol type is contained in the lower order 4 bits. The availble bindings and types are reported in tables 1.2.3 and 1.2.4.

Name	Value	Description
STB_LOCAL	0	Not visible out of the object files
STB_GLOBAL	1	Visible to all object files
STB_WEAK	2	Global but with lower precedence
STB_LOOS	10	Env specific use
STB_HIOS	12	Env specific use
STB_LOPROC	13	Processor specific use
STB_HIPROC	15	Processor specific use

Table 1.2.3: Symbol Bindings

Name	Value	Description
STT_NOTYPE	0	No type
STT_OBJECT	1	Data Object
STT_FUNC	2	Function entry point
STT_SECTION	3	Symbol is associated with a section
STT_FILE	4	Source file associated with the object file
STT_LOOS	10	Env specific use
STT_HIOS	12	Env specific use
STT_LOPROC	13	Processor specific use
STT_HIPROC	15	Processor specific use

Table 1.2.4: Symbol Types

- `st_other` is reserved for future use and must be 0;
- `st_shndx` contains the index of the section the index is defined into. For undefined symbols it is `SHN_UNDEF`, for absolute symbols it is `SHN_ABS` and for common ones is `SHN_COMMON`;

- `st_value` contains the value of the symbol. it may be aboslute or relocatable address. In relocatable files this field contains the alignment constraint for common symbols and a section relative offset for defined relocatable symbol. In executables and standard object files this contains a virtual address for the defined relocatable symbol.

- `st_size` contains the size associated with the symbol. If a symbol does not have an associated size this field contains 0.

1.2.4 Relocations

These sections contain information used by the linker to perform relocations; essentially these are tables with each entry detailing a particular address at whic the relocation must be performed along with the instructions to resolve the value that needs to be plugged at that value. Of course static relocations are resolved at compile time and only dynamic will remain. These sections can be `.rel.*` or `.rela.*` and their type defines the symbol resolution method.

.rel.* The `.rel.*` sections contain the addend part of the relocation from the original part and are smaller than `.rela.*`. The underlying data structure is reported in 1.5.

```
typedef  struct {
  Elf64_Addr r_offset; // Address of reference
  Elf64_Xword r_info;  // Symbol index and type of relocation
} Elf64_Rel;
```

Listing 1.4: .rel.* entries

.rela.* This is larger than the former and provides an explicit field for the full addend.

```
typedef  struct {
  Elf64_Addr r_offset;   // Address of reference
  Elf64_Xword r_info;    // Symbol index and type of relocation
  Elf64_Sxword r_addend; // Constant part of the expression
} Elf64_Rela;
```

Listing 1.5: .rel.* entries

where:

- `r_offset` indicates the location at which the relocation must be applied. For a relocatable file this is the offset, in btes, from the beginning of the section to the beginning of the storage unit being relocated. For an executable or shared object, this is the virtual address of the storage unit being relocated.

- `r_info` contains both a symbol table index and a relocation type. The symbol table index identifies the symbol which value should be used in the relocation. The type instead qualifies the method used for the relocation; these can vary wildly in the real world, but the most important are `R_X86_64_GLOB_DAT` and `R_X86_64_JUMP_SLO`.

  The first one is used to compute the address of a data symbol and plug it in the correct offset into the `.got` section.

  The latter is called jump slot and they have their offset in the `.got.plt` (the stubs in the PLT use them to compute the jump target as an offset from the `rip` register). They can be identified by application of the macro in 1.6.

```
#define ELF64_R_SYM(i)((i) >> 32)
#define ELF64_R_SYM(i)((i) & 0xffffffff)
#define ELF64_R_INFO(s, t)(((s) << 32) + ((t) & 0xffffffff))
```

Listing 1.6: Macros

- `r_addend` specifies a constant addend used to compute the value to be stored in the relocated field.

1.2.5 Program Header Table

The program header table provides the segment view as opposed to the section perspective detailed above and has a different purpose indeed. In fact sections are exclusively relevant to the static linker, while segments are used by the dynamic linker when loading an ELF into a process for execution to locate the relevant code and data and decide what to load in virtual memory. A single segment may contain multiple sections. The underlying data structure can be found in listing 1.7

```
typedef  struct {
  Elf64_Word  p_type;    // Segment type
  Elf64_Word  p_flags;   // Segment attributes
  Elf64_Off   p_offset;  // Constant part of the expression
  Elf64_Addr  p_vaddr;   // Virtual address in memory
  Elf64_Addr  p_paddr;   // Reserved
  Elf64_Xword p_filesz;  // Size of the segment in the file
  Elf64_Xword p_memsz;   // Size of the segment in memory
  Elf64_Xword p_align;   // Alignment of segment
} Elf64_Rela;
```

Listing 1.7: Program Header

where:

- `p_type` indicates the type of the segment and can have the following values: The fields `PT_LOAD`,

Name	Value	Description
PT_NULL	0	Unused entry
PT_LOAD	1	Loadable segment
PT_DYNAMIC	2	Dynamic linking tables
PT_INTERP	3	Program interpreter path
PT_NOTE	4	Note sections
PT_SHLIB	5	Reserved
PT_PHDR	6	Program header table
PT_LOOS	0x60000000	Environment specific use
PT_HIOS	0x6FFFFFFF	Environment specific use
PT_LOPROC	0x70000000	Processor specific use
PT_HIPROC	0x7FFFFFFF	Processor specific use

Table 1.2.5: Segment types

  `PT_DYNAMIC` and `PT_INTERP` are particularly important. `PT_LOAD` segments are intended to be loaded into memory when setting up the process. The size of the chunk at the address which it must be loaded at are described in the rest of the program header. `PT_INTERP` segments contains the `.interp` section which provides the name of the interpreter that is to be used to load the binary. `PT_DYNAMIC` contains the `.dynamic` section which tells the interpreter how to parse and prepare the binary for execution

- `p_flags` contains the segment attributes. The top eight bits are reserved for processor specific use and the others are reserved for environment specific use. The possible attributes can assume the values:

- `p_offset` contains the offset of the segment, in bytes, from the beginning of the file;

- `p_vaddr` contains the virtual address of the segment in memory (for loadable segments this must be equal to the offset);

- `p_paddr` is reserved for the physical address, but in modern OS's is generally set to 0 (most of today's systems use virtual addresses);

Name	Value	Description
PF_X	0x1	Execute permission
PF_W	0x2	Write permission
PF_R	0x4	Read permission
PF_MASKOS	3	Environment reserved
PF_MASKPROC	4	Precessor reserved

Table 1.2.6: Segment types

- p_filesz and p_memsz contain respectively the size of the file image and the memory image of the segment. These are different as some sections only indicate the need to allocate some bytes but don't actually occupy these bytes in the bynary file (e.g. .bss contains zero initialised data and since all data in this section are zero there is no need to actually include them in the binary, but all of its bytes are allocate when it is loaded into memory). Hence it is impossible to have p_memsz bigger than p_filesz and when this happens the loader adds extra bytes initialised to zero.
- p_align specifies the alignment constraints for the segment and must be a powero of two. The p_offset and p_vaddr must be congruent modulo the alignment.

1.2.6 Notes

Sections of type SHT_NOTE and PT_NOTE are used by compilers and other tools to mark an object file with special info that has a special meaning for a particular toolset. These sections and segments contain any number of note entries, each of which is an 8-byte word. A note contains:

- namesz and name. The first identifies the length of a name in bytes. The latter contains a null terminated string, padded to 8-byte alignment.
- descsz and desc. The first identifies the òength of the note descripto whilst the second stores the content of the note padded to 8-byte alignment as necessary.
- type contains a number that determines, along with the oroginaor's name, the interpretation of the note contents.

1.2.7 Dynamic table

Dynamically bound object files will have a PT_DYNAMIC program header entry that refers to a segment contaaining the .dynamic sextion. The content of the section is an array of Elf64_Dyn structures as per the listing below:

```
typedef struct {
      Elf64_Sxword    d_tag;
      union {
          Elf64_Xword d_val;
          Elf64_Addr  d_ptr;
      } d_un;
} Elf64_Dyn;
```

Listing 1.8: Dynamic Table Structure

where:

- d_tag identifies the type of dynamic table entry. The type determiines the interpretation to be given to the union d_un. The processor-independent dynamic table entry types are

Name	Value	d_un	Description
DT_NULL	0	ignored	Marks the end of the dynamic array
DT_NEEDED	1	d_val	The string table offset of a needed library
DT_PLTRELSZ	2	d_val	Total size, in bytes, of the relocation entries associated with the PLT
DT_PLTGOT	3	d_ptr	Address associated with the PLT (the specific meaning is processor dependent)
DT_HASH	4	d_ptr	Address of the symbol hash table
DT_STRTAB	5	d_ptr	Address of the dynamic string table
DT_SYMTAB	6	d_ptr	Address of the dynamic symbol table
DT_RELA	7	d_ptr	Address of the relocation table with Elf64_Rela entries
DT_RELASZ	8	d_val	Total size, in bytes of the DT_RELA table
DT_RELAENT	9	d_ptr	Size, in bytes, of the relocation entry
DT_STRSZ	10	d_val	Total size, in bytes, of the string table
DT_SYMENT	11	d_val	Size, in bytes, of each symbol table entry
DT_INIT	12	d_ptr	Address of the initialisation function
DT_FINI	13	d_ptr	Address of the termination function
DT_SONAME	14	d_val	String table offset of this shared object
DT_RPATH	15	d_val	String table offset of a shared library search path
DT_SYMBOLIC	16	ignored	The presence of this dynamic table entry modifies the symbol resolution algorithm for references within the library. These are resolved befre the dynamic linker searches the usual search path
DT_REL	17	d_ptr	Address of the relocation table with Elf64_Rel entries
DT_RELSZ	18	d_val	Size in bytes of the DT_REL table
DT_RELENT	19	d_val	Size in bytes of the DT_REL relocation entry
DT_PLTREL	20	d_val	Type of the relocation entry for the procedure linkage table. The d_val contains either DT_REL or DT_RELA
DT_DEBUG	21	d_ptr	Reserved for debugger use
DT_TEXTREL	22	ignored	The presence of this dynamic table entry signals that the relocation table contains relocation for non-writable segment
DT_JMPREL	23	d_ptr	Address of the relocations associated with the PLT
DT_BIND_NOW	24	ignored	The presence of this dynamic table entry signals that the dynamic loader should process all relocations before transferring control to th program
DT_INIT_ARRAY	25	d_ptr	Pointer to array of pointers to the initialisation functions
DT_FINI_ARRAY	26	d_ptr	Pointer to array of pointers to the termination functions
DT_INIT_ARRAYSZ	27	d_ptr	Size in bytes of the array of initialisation functions
DT_FINI_ARRAYSZ	28	d_ptr	Size in bytes of the array of termination functions

Table 1.2.7: Dynamic Table Entries

- d_val union member used to represent integer values;
- d_ptr union member used to represent link-time virtual addresses that must be relocated to math the actual address at which the object file is loaded. The relocations must be done implicitly and tthere are no dynamic relocations for these items

1.2.8 Hash Table

The access to the dynamic table is efficiently guaranteed by the use of a hash table that is part of the loaded program segment (typically in the `.hash` section) and is pointed to by the `DT_HASH` entry in the dynamic table. The hash table is an array of `Elf64_Word` objects, organised as:

- `nbucket`
- `nchain`
- `bucket[0]...bucket[nbucket - 1]`
- `chain[0]...chain[nchain - 1]`

The `bucket` array forms the hash table itself. The number of entries in the hash table is given by the first word `nbucket`. The entries in the `chain` reflect the symbol table. Entries of the symbol table are organised in hash chains, one per bucket. The hash function is listed below.

```
unsigned long elf64_hash(const unsigned char *name) {
    unsigned long h = 0, g;
    while (*name) {
        h = (h << 4) + *name++;
        if (g = h & 0xF0000000)
            h ^= g >> 24;
        h &= 0x0FFFFFFF;
    }
    return h;
}
```

Listing 1.9: Program Header

1.3 x86/x64

The x86 is the most widely adopted processor architecture. Started in 1978 with the 8086 and 8088; the processor had segmentation and supported $2^{20} = 1MB$. The 286 was issued in 1982; it had protected mode using segment registers as a selector with a maximum supported address space of $2^{24} = 16MB$. The game changer arrived in 1985 with the 386; it was the first 32-bit processor using virtual 8086 mode with support for an address space of $2^{32} = 4GB$ and paging with 4k pages. The 486 (x87), released in 1989, had an FPU. During the end of the 90's/beginning of the 2000 several revolutions happened:

1993 Pentium (4k and 4M pages);

1995-99 P6 family with support for SIMD instructions and vector (MMX and SSE mainly used for multimedia);

2000-2007 Pentium 4 and Xeon families, enhanced SIMD (SSE2 and 3), hyperthreading and VT. This family sets a milestone as it sees the introduction of the modern IA64

And now we come to 2008 with the release of the contemporary Intel Core i3, i5, i7 family with SSE4.2 and second generation VT.

1.3.1 Modes of operation

The operation modes, also called processor modes, CPU states or privilege level wok basically like gates in the execution of certain instructions. The purpose is to place restrictions on the type and scope of operations that can be performed by certain processes.

32bit

The processor boots up in real-address mode (the 8086 mode) for compatibility. After that it reaches the protected mode where execution is carried out according to the privileges established for the relevant execution ring. There are 4 execution rings from 0 (kernel mode) to 3 (user mode). Applications run with a paged 32-bit flat address space. There is another mode which is the system management mode intended for use by firmware only.

64bit

The 64bit CPUs runs in IA32e, with two submodes:

1. **Compatibility Mode** similar to the 32bit protected mode and permits legacy 32 and 16 bit applications to run without being recompiled. Allows access to $2^{36} = 64GB$ of physical memory using Physical Address Extensions;
2. **64 bit Mode** allows to run 64 bit applications, extending general and SIMD registers from 8 to 16.

Paging and Memory Translation

Paging is the hardware implementation by which multiple programs can use the same full virtual address space without crashing. The translation process starts every time the CPU executes an instruction that refers to a memory address. The origin of this technique dates back to the 8086 that had 16-bit registers and its instructions used mostly 8 or 16 bit operands allowing the code to work with 2^{16} bytes (64K); yet CPU designers wanted to use more memory without expanding the size of the registers. This gave birth to the segment registers which are a mean to tell the CPU which 64K long segment had to be used and the effective offset from the beginning of the segment itself. Originallythere were four registers:

SS used for the stack;

CS used for the code;

DS/ES used for data;

The CPU produces a logical address made of a 16bit Selector and the Offset; these are given to the GDT that returns the Base and the Limit of the LDT along with relevant flags. These are combined into the linear address which contains the Page Directory, the Page Table and the Offset. By knowing the page directory and the page table we can find the PPN and with the offset we finally find the address in memory where data and instructions stand. Modern OS's use paging only to provide a 32/64 bit virtual flat address space.

Registers

In 32 bit mode the segment registers CS, DS, ES and SS hold 16 bit useless segment selectors. In 64 bit CS, DS, ES, SS are all treated as if each segment base is 0 and all limit checks are disabled. FS and GS point to the TLS for 64 and 32 bits respectively. Registers are divided in:

- General Purpose

  RAX/EAX, Accumulator;

  RBX/EBX, Pointer to Data;

  RCX/ECX, Counter;

  RDX/EDX, I/O Pointer;

  RSI/ESI, Source pointer for string operations;

  RDI/EDI, Destination pointer;

  R8-R15, Only for 64 bit;

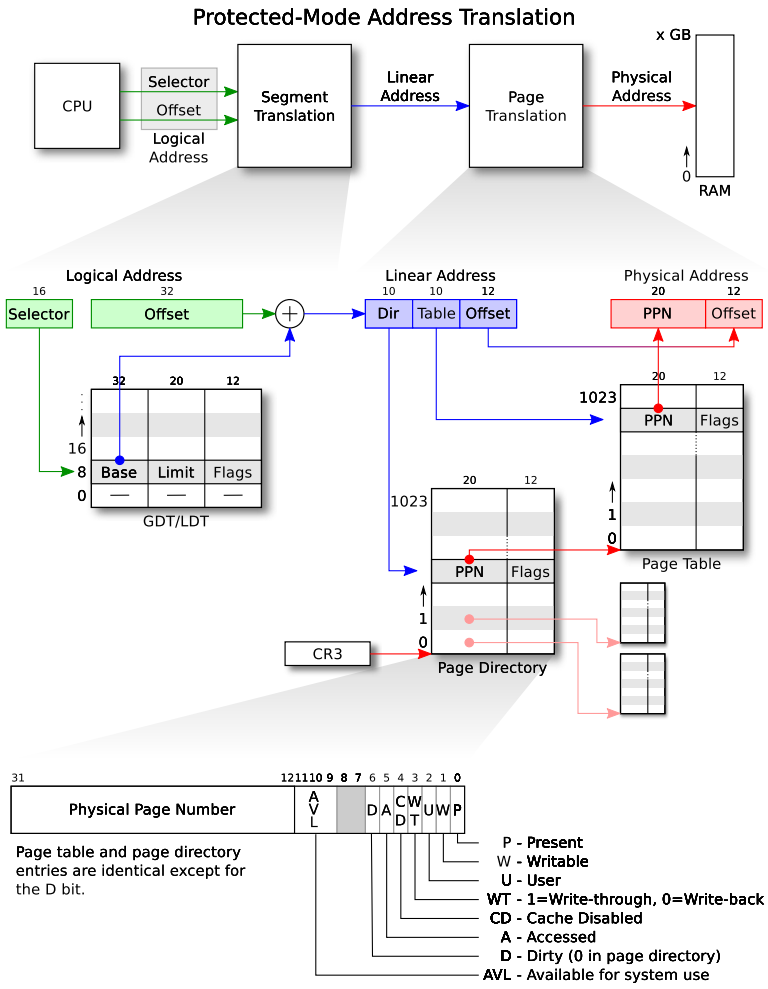

Figure 1.3.1: Memory Translation

- Execution

  RIP/EIP, Instruction pointer;

  RSP/ESP, Stack pointer;

  RBP/EBP, Base pointer;

  RFLAGS/EFLAGS Carry, sign, zero, parity, ...

The instructions can act on zero or more operands and data can be located:

- The instruction itself in case of an immediate operand;
- a register
- a memory location
- an I/O port

Glossary

370-XA System 370 Extended Architecture. 19

ABI Application Binary Interface. 19

API Application Programming Interface. 19

Binary Analysis Analysis of binary computer programs, binaries in short, and the code within them to truly understand their actual behaviour as opposed to their intended behaviour. 19

Dynamic Analysis Analysis of a binary at runtime. 19

ECPS Extended Control Program Support. 19

ELF Executable and Linkable Format. 19

FS File System. 19

GDT Global Description Table. 19

I/O Input/Output. 19

IA-32 Intel Architecture 32 Bit. 19

IA-64 Intel Architecture 64 Bit. 19

ISA Instruction Set Architecture. 19

LDT Local Description Table. 19

OS Operating System. 19

PAE Physical Address Extension. 19

PE Portable Executable. 19

PLT Procedure Linkage Table. 19

PPN Physical Page Number. 19

Static Analysis Analysis of a binary without running it. 19

STB Shadow Table Bypass. 19

TLS Thread Level Storage. 19

VM Virtual Machine. 19

VMA Virtual Machine Assist. 19

VMF Virtual Machine Facility. 19

VMM Virtual Machine Monitor. 19