/*

Last update Sun Dec 22 06:55:39 2002 mayhem

- Version 0.1 May 2001
- Version 0.2 .::. 2002 (WIP) :
  - Added stuff about rtld relocation .
  - Added stuff about rtld symbol resolution .
  - Various fixes and some links added .

This draft remained unreleased for one year, most of it is based on the glibc-2.2.3 implementation, information about the subject has been disclosed on bugtraq and phrack in beg 2002 :

http://online.securityfocus.com/archive/1/274283/2002-05-29/2002-06-04/2
http://www.phrack.org/phrack/59/p59-0x08.txt

However, it still contains some kewl info, I'll try to keep it updated, hope this will help . I am also adding/clearing/correcting stuffs (and giving credits) on demand, so dont hesitate to send comments, etc .

/mM [mayhem at devhell dot org]

*/

Understanding Linux ELF RTLD internals

Most of the paper has been developed in a security perspective, your comments are always welcomed .

Actually there's many ELF documentation at this time, most of them are virii coding or backdooring related . To be honest, I never found any documentation on the dynamic linking sources, and thats why I wrote this one . Sometimes it looks more like an internal ld.so reference or a comments review on the ELF dynamic linking implementation in ld-linux.so .

It's not that unuseful since the dynamic linking is one of the worse documented part of the Linux operating system . I also decided to write a (tiny) chapter on ELF kernel handling code, because it is really necessary to know some kernel level stuffs (like the stack initialisation) to understand the whole interpreting .

You can find the last glibc sources on the GNU's FTP server :

ftp://ftp.gnu.org/pub/gnu/glibc/

If you dont know anything in ELF, you should read the reference before :

http://x86.ddj.com/ftp/manuals/tools/elf.pdf

Want to know more ? Go on !

```
     2] FAQ, thanks and references


TODO    :
                    X) Stack information gathering
                    X) SHT_DYNAMIC information gathering
                    X) PHT interpreting
                    X) Loading shared libraries
                    X) Shared libraries relocation


O] PROLOGUE

   A] KERNEL IMPLEMENTATION


The kernel I worked on is Linux 2.4.4 . I suggest you reading this first
chapter with the kernel sources (/usr/src/linux/fs/binfmt_elf.c !)
meanwhile reading this part of the article .

When a userspace process calls the execve() syscall, the kernel takes control
(int $0x80), the handling code for this interruption is in
/usr/src/linux/arch/i386/kernel/entry.S for i386 architectures .

You can find a big switch statement, providing a way to launch the desired
function giving the %eax value (i.e. the syscall number) . In our case, the
function is sys_execve(), which calls do_execve() .


        sys_execve() in fs/exec.c

                0) open_exec()

                        File opening .

                1) prepare_binprm()

                        Capabilities retreiving .

                3) copy_strings()

                        Environnement and arguments retreiving from userspace
                        to kernelspace . The argv[] strings are recopied .

                4) search_binary_handler()

                        Executable type retreiving .


        search_binary_handler() in fs/exec.c


  In this file, the executable format is scanned in a for loop . The fn()
function pointer is used . When the correct format has been identified
(using the ELF MAGIC COOKIE found in the first bytes of the file), the
related 'linux_binprm' structure is retreived and the load_binary() function
is launched . For the ELF format , this function is load_elf_binary() .

        load_elf_binary() in fs/binfmt_elf.c

                First, there's may concistency checks :
                - check if the processor type is supported (EM_386 or EM_486)
                - check if the file is not too big (more than 65 636 bytes)

                Then we kmalloc() a buffer with the required size (the sum of
```

every segment size : elf_ex.e_phentsize * elf_ex.e_phnum),
and we try to find the PT_INTERP segment (program interpretor,
i.e. the content of the .interp section)

If the interpretor is a ibcs2 one, we set the behavour for
this process, the SET_PERSONALITY() macro is used .

(FIXME: ibcs2 specs)

After that, the interpretor file is opened, the kernel maps
its segments (if there's any problem , the task is killed
using the send_sig() function) . The capabilities for this
processus are set in compute_creds() then some information is
put on the stack in create_elf_tables() .

Here is the stack layout after the create_elf_tables() call :

```
        UP

        0  AT_NULL              /* End of vector */
        0  AT_HWCAP             /* Arch dependent hints */
        1  AT_PAGESZ            /* Page size */
        2  AT_CLKTCK            /* times() incrementation frequency */
        3  AT_PHDR              /* Program header address */
        4  AT_PHENT             /* Program header entry lenght */
        5  AT_PHNUM             /* Program header entries number */
        6  AT_BASE              /* Program base address */
        7  AT_FLAGS             /* Unknown, set to 0 in the kernel */
        8  AT_ENTRY             /* Program entry point */
        9  AT_UID               /* Process uid */
        10 AT_EUID              /* Process euid */
        11 AT_GID               /* Process gid */
        12 AT_EGID              /* Process egid */
ESP=>

        DOWN
```

The entry AT_HWCAP seems strange ... From the kernel sources:

" ... this yields a mask that user programs can use to figure
out what instruction set this CPU supports. This could be
done in user space, but it's not easy, and we've already done
it here .... "

```
#define AT_HWCAP        ELF_HWCAP
#define ELF_HWCAP       (boot_cpu_data.x86_capability[0])

        ...
```

That's not so important to understand the ELF interpreting
so I wont develop here .

After this stack initalizing, memory for the bss section is
allocated with set_brk(), every registers are set to 0 in
the ELF_PLAT_INIT() macro, finally the kernel gives the hand
to the interpretor entry point . For the Linux 2.4.4 kernel
on my SlackWare 7, the _dl_start() function in
/lib/ld-linux.so.2 is called . The kernel start_thread()
macro is used to give the hand to the dynamic linker,
changing the values of the EIP register .

We need now to do the symbol resolution, the relocation,
the whole work is done by the dynamic linker (the
interpretor) . This dynamic linker sources are now included

in the libc (SlackWare 7 glibc version is 2.1.2) .

B] GLIBC MACROS

The glibc developpers have done some macros you can find quite often in the code . The dynamic linker also does some references to these routines .

- weak_alias
- weak_extern
- strong_alias
- _builtin_expect
- link_warning
- stub_warning
- symbol_version
- default_symbol_version

1] DYNAMIC LINKING IMPLEMENTATION

A] SOURCES GRAPH

Here is a simple graph representing the dynamic linker source code hierarchy . The true entry point is _dl_start, this function is called by the kernel from the start_thread() kernel function . At the end of the dynamic linking process, the starting function of the binary , the _start() point, is called .

```
RTLD_START()                                    (sysdeps/i386/dl-machine.h)
  _dl_start()                                   (elf/rtld.c)
        elf_machine_load_addr()
        elf_get_dynamic_info()
        ELF_DYNAMIC_RELOCATE()                  (elf/dynamic-link.h)
          elf_machine_runtime_setup()           (sysdeps/i386/dl-machine.h)
            _ELF_DYNAMIC_DO_RELOC()             (sysdeps/i386/dl-machine.h)
                elf_dynamic_do_rel()            (elf/do-rel.h)
                    elf_machine{,_lazy_}rel()   (sysdeps/i386/dl-machine.h)
  _dl_start_final()                             (elf/rtld.c)
        _dl_sysdep_start()                      (sysdeps/generic/dl-sysdeps.h)
          _dl_main()                            (elf/rtld.c)
             process_envvars()                  (elf/rtld.c)
             elf_get_dynamic_info()
             _dl_setup_hash()                   (elf.dl-lookup.c)
             _dl_new_object()                   (elf/dl-object.c)
             _dl_map_object()                   (elf/dl-load.c)
             _dl_map_object_from_fd()           (elf/dl-load.c)
                add_name_to_object()            (elf/dl-load.c)
                _dl_new_object()                (elf/dl-object.c)
                map_segment()
                ELF_{PREFERED,FIXED}_ADDRESS()
                mprotect()
                munmap()
               _dl_setup_hash()                 (elf/dl-lookup.c)
             _dl_map_object_deps()              (elf/dl-deps.c)
                preload()
                   _dl_lookup_symbol()          (elf/dl-lookup.c)
                      do_lookup()
                _dl_relocate_object()           (loop in elf/dl-reloc.c)
  _start()                                      (main binary)
```

By default, lazy binding is used . The other dynamic linking type is the 'now' binding (everybody may have seen at least one time the RTLD_LAZY and RTLD_NOW macros) . Lazy binding is a bit less performant since the got entry for an external function is resolved each time it's needed . On the contrary, RTLD_NOW binding do every dynamic linking resolutions during process loading , so that the dynamic linker does not need to be called each time you request an access to an external function . You can control this behaviour using the LD_BIND_NOW environement variable . Whereas you use a now or a lazy binding, the function elf_machine_lazy_rel() or the elf_machine_rel() is used .

B] LINK MAP STRUCTURE EXPLAINED

Each object in the dynamic linker is described by a link_map structure . Here are the details . This structure is from elf/link.h, the comments for each field have been cleaned and developed . Use this description more like a reference during the linear code analyze .

```
struct                  link_map
  {

    /* Base address shared object is loaded at.  */
    ElfW(Addr) l_addr;


    /* Absolute file name object was found in.  */
    char *l_name;

    /* Dynamic section of the shared object.  */
    ElfW(Dyn) *l_ld;


    /* Chain of loaded objects.  */
    struct link_map *l_next, *l_prev;


    /*
    ** All following members are internal
    ** to the dynamic linker . They may
    ** change without notice
    */


    /* FIXME */
    struct libname_list *l_libname;


    /* Indexed pointers to dynamic section. */
    ElfW(Dyn) *l_info[DT_NUM +
                      DT_THISPROCNUM +
                      DT_VERSIONTAGNUM +
                      DT_EXTRANUM];


    /* Pointer to program header table in core.  */
    const ElfW(Phdr) *l_phdr;


    /* Entry point location.  */
    ElfW(Addr) l_entry;
```

```
/* Number of program header entries.  */
ElfW(Half) l_phnum;


/* Number of dynamic segment entries.  */
ElfW(Half) l_ldnum;


/*
**   Array of DT_NEEDED dependencies and their dependencies,
**   in dependency order for symbol lookup (with and without
**   duplicates).  There is no entry before the dependencies
**   have been loaded.
*/
struct r_scope_elem l_searchlist;


/*
**  We need a special searchlist to process objects marked
**  with DT_SYMBOLIC.
*/
struct r_scope_elem l_symbolic_searchlist;


/* Dependent object that first caused this object to be loaded. */
struct link_map *l_loader;


/* Symbol hash table.  */
Elf_Symndx l_nbuckets;
const Elf_Symndx *l_buckets, *l_chain;


/* Reference count for dlopen/dlclose.  */
unsigned int l_opencount;


/* Where this object came from.  */
enum
  {
    lt_executable,          /* The main executable program.  */
    lt_library,             /* Library needed by the program */
    lt_loaded               /* Extra runtime loaded shared obj. */
  }
l_type:2;


/* Nonzero if object's relocations done.  */
unsigned int l_relocated:1;


/* Nonzero if DT_INIT function called.  */
unsigned int l_init_called:1;


/* Nonzero if object in _dl_global_scope (FIXME) */
unsigned int l_global:1;


/* Reserved for internal use (FIXME) */
unsigned int l_reserved:2;
```

```
/*
** Nonzero if the data structure pointed to by
** l_phdr is allocated . (FIXME)
*/
unsigned int l_phdr_allocated:1;


/*
** Nonzero if the SONAME is for sure in the
** l_libname list  (FIXME)
*/
unsigned int l_soname_added:1;


/*
** Nonzero if this is a faked descriptor without
** associated file . (FIXME)
*/
unsigned int l_faked:1;


/* Array with version names.  */
unsigned int l_nversions;
struct r_found_version *l_versions;


/* Collected information about own RPATH directories.  (FIXME) */
struct r_search_path_struct l_rpath_dirs;


/* Collected results of relocation while profiling. (FIXME) */
ElfW(Addr) *l_reloc_result;


/* Pointer to the version information if available.  */
ElfW(Versym) *l_versyms;


/* String specifying the path where this object was found.  */
const char *l_origin;


/*
** Start and finish of memory map for this object.
** l_map_start need not be the same as l_addr.
*/
ElfW(Addr) l_map_start, l_map_end;


/*
** This is an array defining the lookup scope for this link map.
** There are at most three different scope lists.
*/
struct r_scope_elem *l_scope[4];


/*
** A similar array, this time only with the local scope.
** This is used occasionally.
*/
struct r_scope_elem *l_local_scope[2];


/*
** This information is kept to check for sure whether a shared
```

```
        ** object is the same as one already loaded.
        */
        dev_t l_dev;
        ino64_t l_ino;


        /* Collected information about own RUNPATH directories.  */
        struct r_search_path_struct l_runpath_dirs;


        /* List of object in order of the init and fini calls.  (FIXME) */
        struct link_map **l_initfini;


        /*
        ** List of the dependencies introduced through symbol binding.
        ** (FIXME)
        */
        unsigned int l_reldepsmax;
        unsigned int l_reldepsact;
        struct link_map **l_reldeps;


        /* Various flag words. (FIXME) */
        ElfW(Word) l_feature_1;
        ElfW(Word) l_flags_1;


        /* Temporarily used in `dl_close'.  (FIXME) */
        unsigned int l_idx;


      };
```

C] INTERPRETOR RELOCATION

The first thing to do is to relocate the interpretor itself, since its a shared library . This is done in _dl_start() . As soon as the rtld is relocated, the _dl_start_final() function is called .

What is done from _dl_start() :

- the 2nd and 3rd entries of the GOT are initialized

Then:

-> STACK INFORMATION GATHERING
-> ENVIRONMENT VARIABLES
-> SHT_DYNAMIC INFORMATION GATHERING
-> PHT INTERPRETING
-> SHARED LIBRARIES LOADING
-> SHARED LIBRARIES REVERSE ORDERED RELOCATION
-> DYNAMIC SYMBOL RESOLUTION

D] RUNTIME GOT PATCHING

The fixup() internal function is called on demand (see elf/dl-runtime.c) . The code use the link_map l_addr field, and read the cached value of the requested symbol in memory . GDB uses this also, maybe we can make a point on something interresting :

```
(...)
 switch (l->l_info[VERSYMIDX (DT_VERSYM)] != NULL)
        {
        default:
          {
            const ElfW(Half) *vernum =
              (const void *) D_PTR (l, l_info[VERSYMIDX (DT_VERSYM)]);
            ElfW(Half) ndx = vernum[ELFW(R_SYM) (reloc->r_info)];
            const struct r_found_version *version = &l->l_versions[ndx];

            if (version->hash != 0)
              {
                result = _dl_lookup_versioned_symbol (strtab + sym->st_name,
                                                      l, &sym, l->l_scope,
                                                      version,
                                                      ELF_MACHINE_JMP_SLOT, 0);
                break;
              }
          }
        case 0:
          result = _dl_lookup_symbol (strtab + sym->st_name, l, &sym,
                                      l->l_scope, ELF_MACHINE_JMP_SLOT, 0);
        }
(...)
```

Trying to hijack functions using version poisonning would be quite fun, however this will be possible only if .got is not already full, so is the .hash poisoning based hijack . In this code, we can also understand how the version entry is found for this symbol : the l_versions[] array of the current link_map entry is indexed with the 2 byte index, this index is found from the symbol relocation entry, easily deduced from the function parameter using l_info[DT_JMPREL], and adding reloc_offset (pushed on stack in .plt) to it .

E] SYMBOL RESOLVING

Lets look at elf/dl-lookup.c :

```
lookup_t
internal_function
_dl_lookup_symbol (const char *undef_name, struct link_map *undef_map,
                   const ElfW(Sym) **ref, struct r_scope_elem *symbol_scope[],
                   int reloc_type, int explicit)
```

There is another version of this function, called _dl_lookup_symbol_skip(), but this second one start looking for the symbol in the objects which were loaded after the current one .

There's nothing difficult to understand there, the rtld is just looking for the good symbol giving its name and hash value, doing various checks and taking care of the binding (GLOBAL, LOCAL, or WEAK) .

This code is also pretty similar to :

```
lookup_t
internal_function
_dl_lookup_versioned_symbol (const char *undef_name,
                             struct link_map *undef_map, const ElfW(Sym) **ref,
                             struct r_scope_elem *symbol_scope[],
                             const struct r_found_version *version,
                             int reloc_type, int explicit)
```

in the same file, but taking 1 arg in +, which is the symbol version .

2] FAQ

This FAQ contains some questions I really wondered during my first step into this code, may be useful ;)

Q) What's the LD.SO starting function which is called by the kernel ?

It's _dl_start() in elf/rtld.c .

Q) I dont understand this Global Offset Table design !

Hehe, here it is :

- the first entry is the address of the .dynamic section for the object
- the second entry is the link_map pointer structure associated with the actual ELF object .
- the third is the address of the runtime mapping function in LD.SO .

Q) What's the runtime GOT fixup function in LD.SO ?

It's the fixup() function in elf/dl-runtime.c .

Q) The ELF documentation says that the .hash section should contain as much entries as symbols . I have problems to retreives my hashes, am I missing something ?

You are probably trying to resolve a symbol in .symtab, .hash only contains entry for dynamic symbol table entries (.dynsym)

Q) Is there a limited number of program headers for a binary ?

As far as I know, there isn't . Note that the authorized types are (from the ld/binutils documentation) :

```
  PT_NULL (0)
Indicates an unused program header.
  PT_LOAD (1)
Indicates that this program header describes a segment to be loaded
from the file.
  PT_DYNAMIC (2)
Indicates a segment where dynamic linking information can be found.
  PT_INTERP (3)
Indicates a segment where the name of the program interpreter may be
found.
  PT_NOTE (4)
Indicates a segment holding note information.
  PT_SHLIB (5)
A reserved program header type, defined but not specified by the
ELF ABI.
  PT_PHDR (6)
Indicates a segment where the program headers may be found.
```

The executable might have no PT_LOAD header entries at all. However, on other implementation like the FreeBSD one dont allow more than 2 PT_LOAD .

Q) Why is the ELF relocation system so sophisticated ?

That's a good question, this dynamic linking system is quite slow, and other executable formats like the PE (win32 Portable Executable) example are good alternatives to the UNIX ELF way . PE only uses an import and an export table . When you import calls (using __imp__ApiName@nn), it looks like a GOT filled at load time (or in runtime, depending on the linking type) . However, you can find some PLT-like mechanism in Portable Executable , if you are using the _ApiName@nn form (thanks to theowl for pointing it out) .

Q) What do the call/jmp offsets means in a relocatable object's code ?

It is not a vaddr, it is a byte index in the module's main symbol table . Remember : relocatable objects dont have a program header table, neither absolute addressing .

Q) What's that HLT x86 instruction ?!

From the INTEL documentation : The HLT instruction put the processor in sleep mode until an interupt occurs . If you look at the _exit() code in the libc, you can see that it calls the abort() function (the default signal handler for SIGABRT) . The function ends with a :

```
        while (1)
          __asm__("hlt");
```

If you try to execute this from ring 3 (userland process), you would be killed since hlt is privilegied instruction . That's why I think this loop is just the last hope to stop the process :>

Questions ? Bug Reports ? Just to say hello ? mail me !

EOF