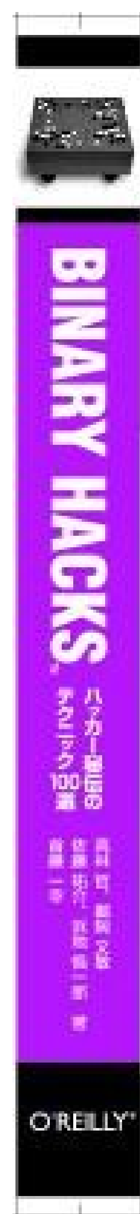
プログラムがmainにたどりつくまで

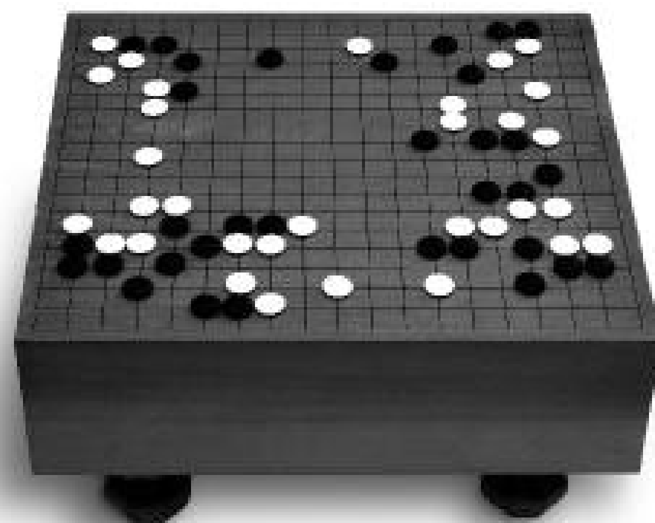
第8回
オープンソース
テクノロジー勉強会

2006/10/24
鵜飼文敏
Debian Developer



BINARY HACKS™

ハッカー秘伝のテクニック100選



高林 哲、鵜飼 文敏
佐藤 祐介、浜地 慎一郎 著
首藤 一幸

O'REILLY®
オライリー・ジャパン

発表の概要

- 「Binary Hacks」のなかから1ハック
  - プログラムがmainにたどりつくまで
    - Code Reading
      - 基本ソフトウェアのコードを読みとく
    - Binary Hacks
      - 基本ソフトウェアの隠された機能を調べる
    - ➔ 性能
    - ➔ 信頼性
    - ➔ 不可思議な現象の解決

Hello, World

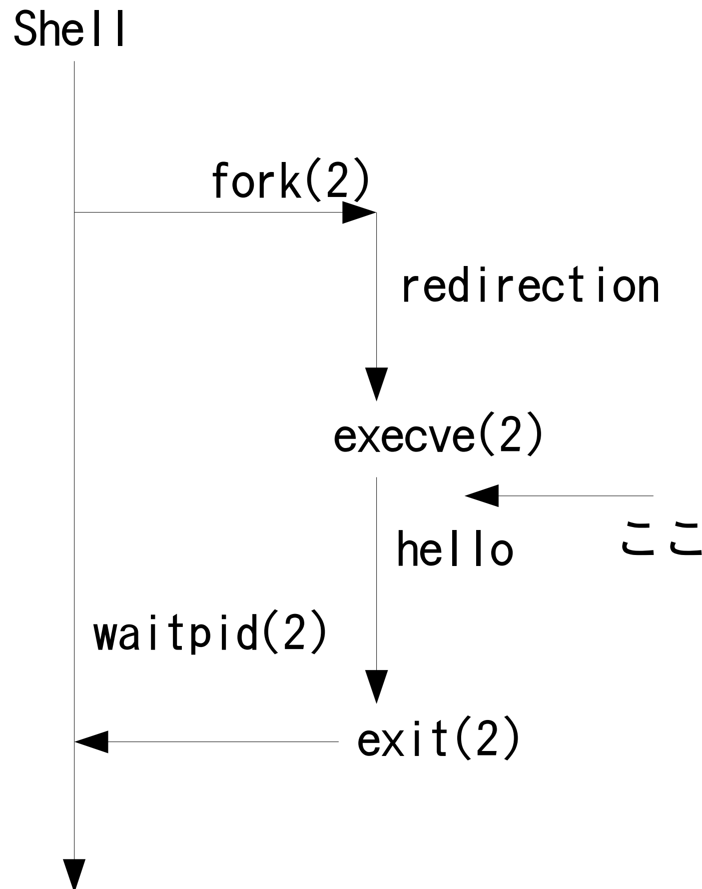
```
#include <stdio.h> /* printf() */
#include <stdlib.h> /* exit() */

int
main(int argc, char **argv)
{
    printf( "Hello, world\n" );
    exit(0);
}
```

```
% cc -o hello hello.c
% ./hello
Hello, world
%
```

ここ

- helloプログラムを実行するとmain()から処理がおこなわれる。
- shellがプロセスを生成してhelloを実行開始してmain()にくるまで。この行間をよむ。

Linuxのプロセス実行



- fork(2)して新しいプロセスを作る
- リダイレクト等の前処理
- exec(2)でhelloプログラムにかわる
  - helloのmainから実行
- exit(2)するとwaitpid(2)しているシェルに終了通知

execve

```
int execve(const char *filename,
           char *const argv[],
           char *const envp[]);
```

- 実行ファイルをメモリーにロード
- エントリーポイントから実行

- execve(2)呼びだし
  - 実行ファイルの指定
  - 引数パラメータ
  - 環境変数

実行ファイル

```
% file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux
  2.2.0, dynamically linked (uses shared libs), not stripped
```

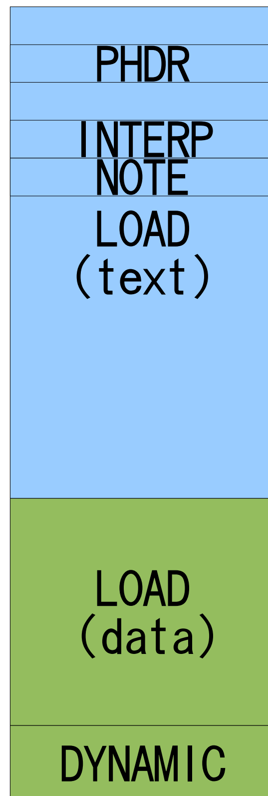
- ELF: Executable and Linking Format
- LSB: Least significant byte first
  - little endian

実行ファイル

```
% readelf -h -l hello
ELF ヘッダ:
  マジック:  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  クラス:                            ELF32
  データ:                            2 の補数、リトルエンディアン
  バージョン:                        1 (current)
  OS/ABI:                            UNIX - System V
  ABI バージョン:                    0
  タイプ:                            EXEC (実行可能ファイル)
  マシン:                            Intel 80386
  バージョン:                        0x1
  エントリポイントアドレス:          0x80482e0
  プログラムの開始ヘッダ:            52 (バイト)
  セクションヘッダ始点:              3672 (バイト)
  フラグ:                            0x0
  このヘッダのサイズ:                52 (バイト)
  プログラムヘッダサイズ:            32 (バイト)
  プログラムヘッダ数:                7
  セクションヘッダ:                  40 (バイト)
  Number of section headers:         34
  Section header string table index: 31
```

実行ファイル

```
Program Headers:
  タイプ       オフセット 仮想Addr   物理Addr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
      [要求されるプログラムインタプリタ: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004cc 0x004cc R E 0x1000
  LOAD           0x0004cc 0x080494cc 0x080494cc 0x00108 0x0010c RW  0x1000
  DYNAMIC        0x0004e0 0x080494e0 0x080494e0 0x000c8 0x000c8 RW  0x4
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

 セグメントマッピングへのセクション:
  セグメントセクション...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag
   06
```



.interp

.note.ABI-tag

.hash .dynsym .dynstr .gnu.version .gnu.version_r
.rel.dyn .rel.plt .iint .plt .text .fini .rodata
.eh_frame

.ctors .dtros .jcr .got .got.plt .data .bss

.dynamic

実行ファイル

```
% objdump -f -p hello

hello:     ファイル形式 elf32-i386
アーキテクチャ: i386, フラグ 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
開始アドレス 0x080482e0

プログラムヘッダ:
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x000000e0 memsz 0x000000e0 flags r-x
  INTERP off    0x00000114 vaddr 0x08048114 paddr 0x08048114 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x000004cc memsz 0x000004cc flags r-x
    LOAD off    0x000004cc vaddr 0x080494cc paddr 0x080494cc align 2**12
         filesz 0x00000108 memsz 0x0000010c flags rw-
 DYNAMIC off    0x000004e0 vaddr 0x080494e0 paddr 0x080494e0 align 2**2
         filesz 0x000000c8 memsz 0x000000c8 flags rw-
    NOTE off    0x00000128 vaddr 0x08048128 paddr 0x08048128 align 2**2
         filesz 0x00000020 memsz 0x00000020 flags r--
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rw-
```

共有ライブラリー

```
% ldd hello
        libc.so.6 => /lib/libc.so.6 (0x4001c000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

- lddを使うとどの共有ライブラリーを使うかわかる。

execve呼びだし

- glibc sysdeps/unix/sysv/linux/execve.c

```
int
__execve (file, argv, envp)
 const char *file;
 char *const argv[];
 char *const envp[];
{
 return INLINE_SYSCALL (execve, 3, file, argv, envp);
}
```

INLINE_SYSCALL

- glibc sysdeps/unix/sysv/linux/i386/sysdep.h

```
#define INLINE_SYSCALL(name, nr, args...) \
  ({                                                          \
    unsigned int resultvar = INTERNAL_SYSCALL (name, , nr, args);     \
    if (__builtin_expect (INTERNAL_SYSCALL_ERROR_P (resultvar, ), 0))  \
      {                                                       \
        __set_errno (INTERNAL_SYSCALL_ERRNO (resultvar, ));         \
        resultvar = 0xffffffff;                               \
      }                                                       \
    (int) resultvar; })
```

INTERNAL_SYSCALL

- glibc sysdeps/unix/sysv/linux/i386/sysdep.h

```
# define INTERNAL_SYSCALL(name, err, nr, args...) \
  ({                                                                    \
    register unsigned int resultvar;                                    \
    EXTRAVAR_##nr                                                       \
    asm volatile (                                                      \
    LOADARGS_##nr                                                       \
    "movl %1, %%eax\n\t"                                                \
    "int $0x80\n\t"                                                     \
    RESTOREARGS_##nr                                                    \
    : "=a" (resultvar)                                                  \
    : "i" (__NR_##name) ASMFMT_##nr(args) : "memory", "cc");            \
    (int) resultvar; })

# define ASMFMT_3(arg1, arg2, arg3) \
        , "b" (arg1), "c" (arg2), "d" (arg3)
```

execve呼びだし

```
movl <envp>,%edx
movl <argv>,%ecx
movl <file>,%ebx
movl $11,%eax     # execve
int $0x80
```

`int $0x80`でkernel modeに遷移

system call

movl $11,%eax

int $0x80

ソフトウェア割り込み
SYSCALL_VECTORが0x80

SYSCALL_VECTOR → system_call:

boot時にset_system_gate()で
SYSCALL_VECTORへの割り込み時の飛び先が
system_callに設定されている

```
        :
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
```

```
sys_call_table:
     :
     :
   .long SYMBOL_NAME(sys_execve)
```

%eaxがsys_call_table内のindex

sys_execve

- linux arch/i386/kernel/process.c

```
asmlinkage int sys_execve(struct pt_regs regs)
{
	int error;
	char * filename;

	filename = getname((char *) regs.ebx);
	error = PTR_ERR(filename);
	if (IS_ERR(filename))
		goto out;
	error = do_execve(filename, (char **) regs.ecx, (char **) regs.edx, &regs);
	if (error == 0)
		current->ptrace &= ~PT_DTRACE;
	putname(filename);
out:
	return error;
}
```

do_execve

- linux fs/exec.c

```
int do_execve(char * filename,
    char __user *__user *argv,
    char __user *__user *envp,
    struct pt_regs * regs)
{
        struct linux_binprm *bprm;
        struct file *file;
            :
        file = open_exec(filename);
             :
        bprm->file = file;
        bprm->filename = filename;
             :
        retval = search_binary_handler(bprm,regs);
        if (retval >= 0) {
            /* execve success */
                   :
            return retval;
        }
}
```

search_binary_handler

- linux fs/exec.c

```
/*
 * cycle the list of binary formats handler, until one recognizes the image
 */
int search_binary_handler(struct linux_binprm *bprm,struct pt_regs *regs)
{
    :
      for (fmt = formats ; fmt ; fmt = fmt->next) {
       int (*fn)(struct linux_binprm *, struct pt_regs *) = fmt->load_binary;
       :
       retval = fn(bprm, regs);
       :
      }
   :
}
```

struct linux_binfmt

- linux fs/binfmt_elf.c

```
static struct linux_binfmt elf_format = {
                .module         = THIS_MODULE,
                .load_binary    = load_elf_binary,
                .load_shlib     = load_elf_library,
                .core_dump      = elf_core_dump,
                .min_coredump   = ELF_EXEC_PAGESIZE
};
```

load_elf_binary

- linux fs/binfmt_elf.c

```
static int load_elf_binary(struct linux_binprm * bprm, struct pt_regs * regs)
{
        :
   /* Get the exec-header */
   loc->elf_ex = *((struct elfhdr *) bprm->buf);
         :
   /* First of all, some simple consistency checks */
   if (memcmp(loc->elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
      goto out;
         :
   retval = kernel_read(bprm->file, loc->elf_ex.e_phoff, (char *) elf_phdata, size);
         :
out:
          :
}
```

load_elf_binary

- linux fs/binfmt_elf.c

```
for (i = 0; i < loc->elf_ex.e_phnum; i++) {
  if (elf_ppnt->p_type == PT_INTERP) {
    /* This is the program interpreter used for
     * shared libraries - for now assume that this
     * is an a.out format binary
     */
      :
    retval = kernel_read(bprm->file, elf_ppnt->p_offset,
                         elf_interpreter,
                         elf_ppnt->p_filesz);
                :
    interpreter = open_exec(elf_interpreter);
    retval = PTR_ERR(interpreter);
    if (IS_ERR(interpreter))
      goto out_free_interp;
    retval = kernel_read(interpreter, 0, bprm->buf, BINPRM_BUF_SIZE);
  }
}
```

ELF program header

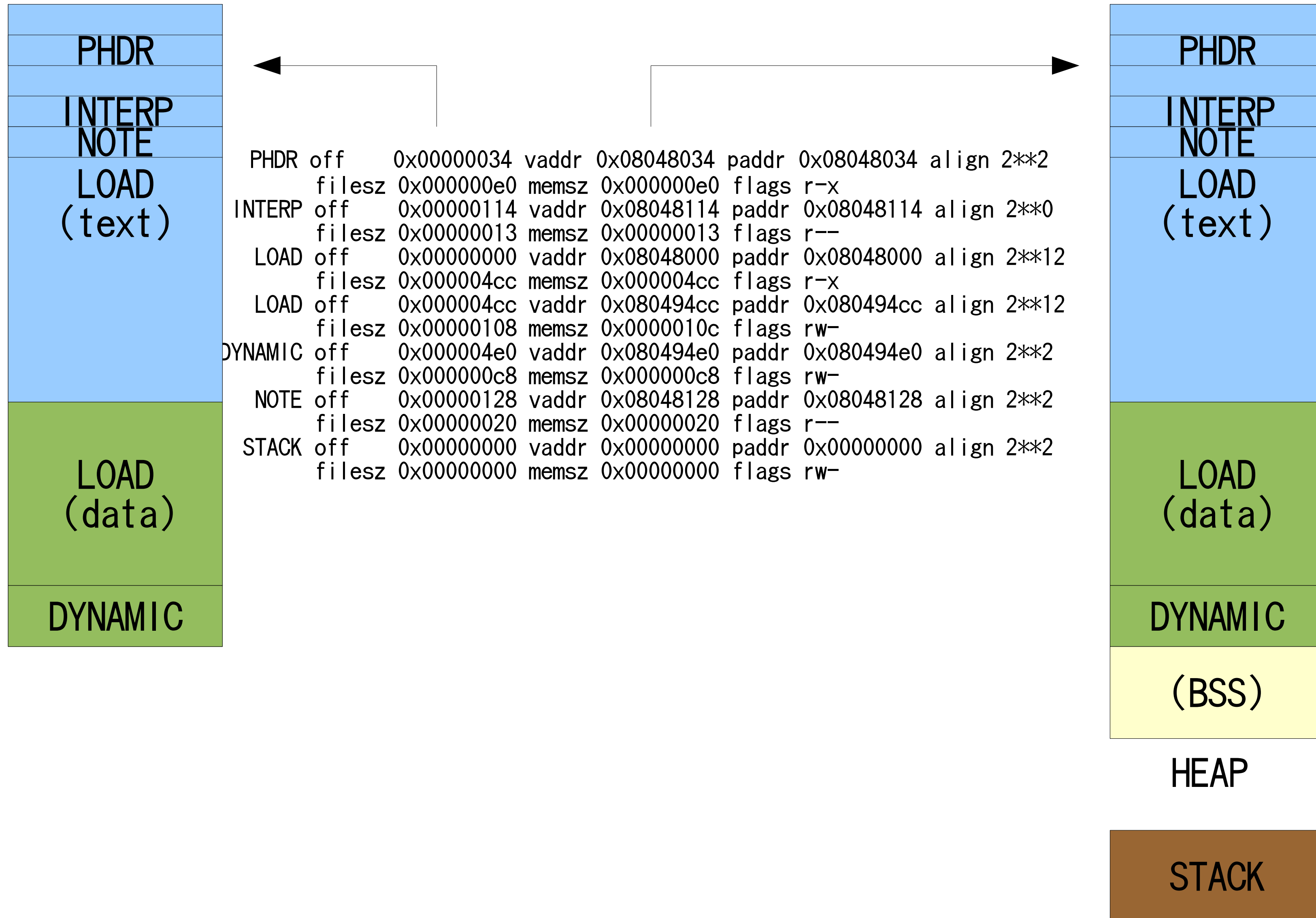
```
% objdump -p hello

hello:     ファイル形式 elf32-i386

プログラムヘッダ:
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x000000e0 memsz 0x000000e0 flags r-x
  INTERP off    0x00000114 vaddr 0x08048114 paddr 0x08048114 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x000004cc memsz 0x000004cc flags r-x
    LOAD off    0x000004cc vaddr 0x080494cc paddr 0x080494cc align 2**12
         filesz 0x00000108 memsz 0x0000010c flags rw-
 DYNAMIC off    0x000004e0 vaddr 0x080494e0 paddr 0x080494e0 align 2**2
         filesz 0x000000c8 memsz 0x000000c8 flags rw-
    NOTE off    0x00000128 vaddr 0x08048128 paddr 0x08048128 align 2**2
         filesz 0x00000020 memsz 0x00000020 flags r--
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rw-
```

```
% od -t x1z -j 0x114 -N 0x13 hello
0000424 2f 6c 69 62 2f 6c 64 2d 6c 69 6e 75 78 2e 73 6f  >/lib/ld-linux.so<
0000444 2e 32 00                                         >.2.<
0000447
```

load_elf_binary

```
/* Flush all traces of the currently running executable */
retval = flush_old_exec(bprm);
  :
/* OK, This is the point of no return */
current->mm->start_data = 0;
current->mm->end_data = 0;
current->mm->end_code = 0;
current->mm->mmap = NULL;
current->flags &= ~PF_FORKNOEXEC;
current->mm->def_flags = def_flags;
  :
/* Now we do a little grungy work by mmaping the ELF image into
   the correct location in memory.  At this point, we assume that
   the image should be loaded at fixed address, not at a variable
   address. */
for(i = 0, elf_ppnt = elf_phdata; i < loc->elf_ex.e_phnum; i++, elf_ppnt++) {
      :
   if (elf_ppnt->p_type != PT_LOAD)
      continue;
      :
   error = elf_map(bprm->file, load_bias + vaddr, elf_ppnt, elf_prot, elf_flags);
      :
}
```

ELFのload



load_elf_binary

```
if (elf_interpreter) {
        :
                elf_entry = load_elf_interp(&loc->interp_elf_ex,
                                            interpreter,
                                           &interp_load_addr);
        :
}

 :

start_thread(regs, elf_entry, bprm->p);
```

start_thread

- linux include/asm-i386/processor.h

```
#define start_thread(regs, new_eip, new_esp) do {              \
        __asm__("movl %0,%%fs ; movl %0,%%gs": :"r" (0));      \
        set_fs(USER_DS);                                       \
        regs->xds = __USER_DS;                                 \
        regs->xes = __USER_DS;                                 \
        regs->xss = __USER_DS;                                 \
        regs->xcs = __USER_CS;                                 \
        regs->eip = new_eip;                                   \
        regs->esp = new_esp;                                   \
} while (0)
```

return from system call

- linux arch/i386/kernel/entry.S

```
        movl %eax,EAX(%esp)              # save the return value
ENTRY(ret_from_sys_call)
        cli                              # need_resched and signals atomic test
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
restore_all:
        RESTORE_ALL
```

ELFインタプリタの実行

- glibc sysdeps/i386/dl-machine.h

```
/* Initial entry point code for the dynamic linker.
   The C function `_dl_start' is the real entry point;
   its return value is the user program's entry point.  */

#define RTLD_START asm ("\n\
        .text\n\
        .align 16\n\
0:      movl (%esp), %ebx\n\
        ret\n\
        .align 16\n\
.globl _start\n\
.globl _dl_start_user\n\
_start:\n\
        # Note that _dl_start gets the parameter in %eax.\n\
        movl %esp, %eax\n\
        call _dl_start\n\
```

dl_start()

- glibc elf/rtld.c

```
static ElfW(Addr) __attribute_used__ internal_function
_dl_start (void *arg)
{
  :
  /* Figure out the run-time load address of the dynamic linker itself.  */
  bootstrap_map.l_addr = elf_machine_load_address ();

  :
  /* Call the OS-dependent function to set up life so we can do things like
     file access.  It will call 'dl_main' (below) to do all the real work
     of the dynamic linker, and then unwind our frame and run the user
     entry point on the same stack we entered on.  */
  start_addr = _dl_sysdep_start (arg, &dl_main);
```

dl_main()

- glibc elf/rtld.c

```
static void
dl_main (const ElfW(Phdr) *phdr,
         ElfW(Word) phnum,
         ElfW(Addr) *user_entry)
{
  :
  /* Process the environment variable which control the behaviour.  */
  process_envvars (&mode);


}
```

環境変数

LD_WARN　警告レベル。出力するかしないか
LD_DEBUG　ダイナミックリンカのデバッグ
LD_VERBOSE　バージョン情報を出力する
LD_PRELOAD　プリロードする共有オブジェクトを指定する
LD_PROFILE　プロファイルをとる共有オブジェクトを指定する
LD_BIND_NOW　遅延バインドをしない
LD_BIND_NOT　ダイナミックリンカではバインドしない
LD_SHOW_AUXV　カーネルからわたらせる補助情報を表示する
LD_HWCAP_MASK　ハードウェアケーパビリティのマスクを設定する
LD_ORIGIN_PATH　バイナリを見つけたパスを設定する
LD_LIBRARY_PATH　共有ライブラリの検索パスを設定する
LD_DEBUG_OUTPUT　デバッグ出力するファイル名を設定する
LD_DYNAMIC_WEAK　weakシンボルも使う
LD_PROFILE_OUTPUT　プロファイル出力するファイル名を設定する
LD_TRACE_PRELINKING　プレリンクをトレースする
LD_TRACE_LOADED_OBJECTS　ロードした共有オブジェクトをトレースする

LD_DEBUG

```
% LD_DEBUG=help /lib/ld-2.3.6.so
Valid options for the LD_DEBUG environment variable are:

 libs        display library search paths
 reloc       display relocation processing
 files       display progress for input file
 symbols     display symbol table processing
 bindings    display information about symbol binding
 versions    display version dependencies
 all         all previous options combined
 statistics  display relocation statistics
 unused      determined unused DSOs
 help        display this help message and exit

To direct the debugging output into a file instead of standard output
a filename can be specified using the LD_DEBUG_OUTPUT environment variable.
```

/lib/ld-2.3.6.so

```
% /lib/ld-2.3.6.so
Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...]
You have invoked `ld.so', the helper program for shared library executables.
This program usually lives in the file `/lib/ld.so', and special directives
in executable files using ELF shared libraries tell the system's program
loader to load the helper program from this file.  This helper program loads
the shared libraries needed by the program executable, prepares the program
to run, and runs it.  You may invoke this helper program directly from the
command line to load and run an ELF executable file; this is like executing
that file itself, but always uses this helper program from the file you
specified, instead of the helper program file specified in the executable
file you run.  This is mostly of use for maintainers to test new versions
of this helper program; chances are you did not intend to run this program.

  --list                list all dependencies and how they are resolved
  --verify              verify that given object really is a dynamically linked
                        object we can handle
  --library-path PATH   use given PATH instead of content of the environment
                        variable LD_LIBRARY_PATH
  --inhibit-rpath LIST  ignore RUNPATH and RPATH information in object names
                        in LIST
```

共有オブジェクトのロード（preload）

```
 if (__builtin_expect (preloadlist != NULL, 0))
   {
     /* The LD_PRELOAD environment variable gives list of libraries
        separated by white space or colons that are loaded before the
        executable's dependencies and prepended to the global scope
        list.  If the binary is running setuid all elements
        containing a '/' are ignored since it is insecure.  */
    :
   }

   :

/* There usually is no ld.so.preload file, it should only be used
    for emergencies and testing.  So the open call etc should usually
    fail.  Using access() on a non-existing file is faster than using
    open().  So we do this first.  If it succeeds we do almost twice
    the work but this does not matter, since it is not for production
    use.  */
 static const char preload_file[] = "/etc/ld.so.preload";
 if (__builtin_expect (__access (preload_file, R_OK) == 0, 0))
   {
         :
   }
```

共有オブジェクトのロード（DT_NEEDED）

```
/* Load all the libraries specified by DT_NEEDED entries.  If LD_PRELOAD
   specified some libraries to load, these are inserted before the actual
   dependencies in the executable's searchlist for symbol resolution.  */
  :
_dl_map_object_deps (main_map, preloads, npreloads, mode == trace, 0);
  :
```

relocation

- glibc elf/dynamic-link.h

```
/* This can't just be an inline function because GCC is too dumb
   to inline functions containing inlines themselves.  */
# define ELF_DYNAMIC_RELOCATE(map, lazy, consider_profile) \
  do {                                                                  \
    int edr_lazy = elf_machine_runtime_setup ((map), (lazy),           \
                                              (consider_profile));      \
    ELF_DYNAMIC_DO_REL ((map), edr_lazy);                               \
    ELF_DYNAMIC_DO_RELA ((map), edr_lazy);                              \
  } while (0)
```

- glibc sysdeps/i386/dl-machine.h

```
/* Set up the loaded object described by L so its unrelocated PLT
   entries will jump to the on-demand fixup code in dl-runtime.c.  */
static inline int __attribute__ ((unused, always_inline))
elf_machine_runtime_setup (struct link_map *l, int lazy, int profile)
{  :
  got[1] = (Elf32_Addr) l;  /* Identify this shared object.  */
   :
  got[2] = (Elf32_Addr) &_dl_runtime_resolve;
   :
}
```

PLTとGOT

```
.text
        ....
        call PLT[n]への相対アドレス
        ....
.plt:

    PLT[n]:
        jmp     *GOT[n]
    PLT_resolv[n]:
        push エントリインデックス
        jmp     レゾルバ

レゾルバ
        pushl    GOT[1]
        jmp  *GOT[2]

.got:     ...
        GOT[1] この共有オブジェクト自体を識別するための情報
        GOT[2] dl_runtime_resolve
            :
        GOT[n] PLT_resolv[n]
```

dl_runtime_resolveが更新

本来のサブルーチンのアドレス

ELFバイナリのエントリ

- glibc sysdeps/i386/elf/start.S (crt1.o)

```
        .text
        .globl _start
        .type _start,@function
_start:
        /* Clear the frame pointer.  The ABI suggests this be done, to mark
           the outermost frame obviously.  */
        xorl %ebp, %ebp

        /* Extract the arguments as encoded on the stack and set up
           the arguments for `main': argc, argv.  envp will be determined
           later in __libc_start_main.  */
        popl %esi               /* Pop the argument count.  */
        movl %esp, %ecx         /* argv starts just at the current stack top.*/

          :
        pushl %edx              /* Push address of the shared library
                                   termination function.  */
```

ELFバイナリのエントリ(cont.)

```
        /* Load PIC register.  */
        call 1f
        addl $_GLOBAL_OFFSET_TABLE_, %ebx

        /* Push address of our own entry points to .fini and .init.  */
        leal __libc_csu_fini@GOTOFF(%ebx), %eax
        pushl %eax
        leal __libc_csu_init@GOTOFF(%ebx), %eax
        pushl %eax
        pushl %ecx              /* Push second argument: argv.  */
        pushl %esi              /* Push first argument: argc.  */
        pushl BP_SYM (main)@GOT(%ebx)

        /* Call the user's main function, and exit with its value.
           But let the libc call main.    */
        call BP_SYM (__libc_start_main)@PLT

1:      movl    (%esp), %ebx
        ret
```

__libc_start_main

- glibc sysdeps/generic/libc-start.c

```
STATIC int
LIBC_START_MAIN (int (*main) (int, char **, char ** MAIN_AUXVEC_DECL),
                 int argc, char *__unbounded *__unbounded ubp_av,
                 void (*init) (void),
                 void (*fini) (void),
                 void (*rtld_fini) (void), void *__unbounded stack_end)
{
  :
  /* Register the destructor of the dynamic linker if there is any. */
  if (__builtin_expect (rtld_fini != NULL), 1))
    __cxa_atexit ((void (*) (void *)) rtld_fini, NULL, NULL);
  :
  /* Register the destructor of the program, if any. */
  if (fini)
    __cxa_atexit ((void (*) (void *)) fini, NULL, NULL);
   :
  /* Call the initializer of the program, if any. */
  if (init)
    (*init) (
#ifdef INIT_MAIN_ARGS
          args, argv, __environ MAIN_AUXVEC_PARAM
#endif
          );
  :
  result = main (argc, argv, __environ MAIN_AUXVEC_PARAM);
  :
  exit (result);
}
```

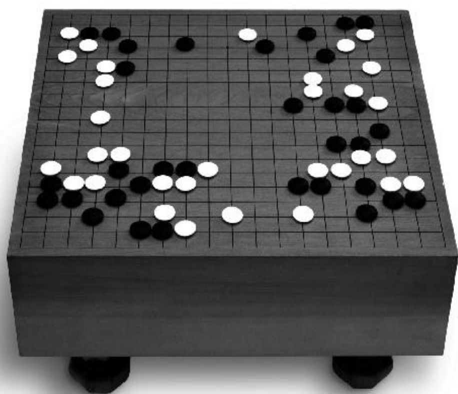
まとめ

- kernel
  - execveで指定されたbinaryとそのELF interpreterをmmap
  - ELF interpreterの entryから実行を開始させる
- ELF interpreter
  - binaryのELFヘッダをみて必要なshared libsのロード、relocationをおこなう
  - 環境変数LD_*でいろいろdebugに便利な情報が得られる。
  - binaryのentryから実行
- binaryのentryはcrt1.o。
  - この中で_initの実行、mainの実行、終了時に_finiを実行。

来月発売!

BINARY HACKS™

ハッカー秘伝のテクニック100選



高林 哲、鵜飼 文敏
佐藤 祐介、浜地 慎一郎 著
首藤 一幸

O'REILLY®
オライリー・ジャパン

抽象化によって隠された
計算機の本当の力を引き出す
プログラミングのノウハウが満載!

基本ツールの使い方、GCCの拡張機能、
セキュアプログラミング、さらにOSの
システムコールやインラインアセンブラを
駆使した高度なテクニックまで解説。
名うてのハッカーたちがこの一冊のために
集結し、低レイヤのプログラミング技術の
活用法とディープな楽しみ方を明らかに
する。