This paper is going to shed some insights into the more obscure security weaknesses of statically linked executables: the glibc initialization process, what the attack surface looks like, and why the security mitigation known as RELRO is as equally important for static executables as it is for dynamic executables. We will discuss some solutions, and explore the experimental software that I have presented as a solution for enabling RELRO binaries that are statically linked, usually to avoid complex dependency issues. We will also take a look at ASLR, and innovate a solution for making it work on statically linked executables.

Standard ELF Security Mitigations

Over the years there have been some innovative and progressive overhauls that have been incorporated into glibc, the linker, and the dynamic linker, in order to make certain security mitigations possible. Firstly there was Pipacs who decided that making ELF programs that would otherwise be ET_EXEC (executables) could benefit from becoming ET_DYN objects, which are shared libraries. If a PT_INTERP segment is added to an ET_DYN object to specify an interpreter then ET_DYN objects can be linked as executable programs which are position independent executables, "-fPIC -pie" and linked with an address space that begins at 0x0. This type of executable has no real absolute address space until it has been relocated into a randomized address space by the kernel. A PIE executable uses IP relative addressing mode so that it can avoid using absolute addresses; consequently, a program that is an ELF ET_DYN can make full use of ASLR.

(ASLR can work with ET_EXEC’s with PaX using a technique called VMA mirroring, but I can’t say for sure if its still supported and it was never the preferred method.)

When an executable runs privileged, such as sshd, it would ideally be compiled and linked into a PIE executable which allows for runtime relocation to a random address space, thus hardening the attack surface into far more hostile playing grounds.

Try running `readelf -e /usr/sbin/sshd | grep DYN` and you will see that it is (most likely) built this way.

Somewhere along the way came RELRO (read-only relocations) a security mitigation technique that has two modes: partial and full. By default only the partial relro is enforced because full-relro requires strict linking which has less efficient program loading time due to the dynamic linker binding/relocating immediately (strict) vs. lazy. But full RELRO can be very powerful for hardening the attack surface by marking specific areas in the data segment as read-only. Specifically the .init_array, .fini_array, .jcr, .got, .got.plt sections. The .got.plt section and .fini_array are the most frequent targets for attackers since these contain function pointers into shared library routines and destructor routines, respectively.

What about static linking?

Developers like statically linked executables because they are easier to manage, debug, and ship; everything is self contained. The chances of a user running into issues with a statically linked executable are far less than with a dynamically linked executable which require dependencies, sometimes hundreds of them. I’ve been aware of this for some time, but I was remiss to think that statically linked executables don’t suffer from the same ELF security problems as dynamically linked executables! To my surprise, a statically linked executable is vulnerable to many of the same attacks as a dynamically linked executable, including shared library injection, .dtors (.fini_array) poisoning, and PLT/GOT poisoning.

This might surprise you; shouldn’t a static executable be immune to relocation table tricks? Let’s start with shared library injection. A shared library can be injected into the process address space using ptrace injected shellcode for malware purposes, however if full RELRO is enabled coupled with PaX improtect restrictions this becomes impossible since the PaX feature prevents the default behavior of allowing ptrace to write to read-only segments and full RELRO would ensure read-only protections on the relevant data segment areas. Now, from an exploitation standpoint this becomes more interest-

---

29 VMA Mirroring by PaX Team: unzip pocorgtf018.pdf vmmirror.txt
ing when you realize that the PLT/GOT is still a thing in statically linked executables, and we will discuss it shortly, but in the meantime just know that the PLT/GOT contains function pointers to libc routines. The .init_array/.fini_array function pointers respectively point to initialization and destructor routines. Specifically .dtors has been used to achieve code execution in many types of exploits, although I doubt its abuse is ubiquitous as the .got.plt section itself. Let’s take a tour of a statically linked executable and analyze the finer points of the security mitigations—both present and absent—that should be considered before choosing to statically link a program that is sensitive or runs privileged.

Demystifying the Ambiguous

The static binary in Figure 17 was built with full RELRO flags, gcc -static -Wl,-z,relro,-z,now. And even the savvy reverser might be fooled into thinking that RELRO is in-fact enabled. partial-RELRO and full-RELRO are both incompatible with statically compiled binaries at this point in time, because the dynamic linker is responsible for re-mapping and mprotecting the common attack points within the data segment, such as the PLT/GOT, and as shown in Figure 17 there is no PT_INTERP to specify an interpreter nor would we expect to see one in a statically linked binary. The default linker script is what directs the linker to create the GNU_RELRO segment, even though it serves no current purpose.

Notice that the GNU_RELRO segment points to the beginning of the data segment which is usually where you would want the dynamic linker to mprotect n bytes as read-only. however, we really don’t want .tdata marked as read-only, as that will prevent multi-threaded applications from working.

So this is just another indication that the statically built binary does not actually have any plans to enable RELRO on itself. Alas, it really should, as the PLT/GOT and other areas such as .fini_array are as vulnerable as ever. A common tool named checksec.sh uses the GNU_RELRO segment as one of the markers to denote whether or not RELRO is enabled on a binary, and in the case of statically compiled binaries it will report that partial-relro is enabled, because it cannot find a DT_BIND_NOW dynamic segment flag since there are no dynamic segments in statically linked executables. Let’s take a lightweight tour through the init code of a statically compiled executable.

From the output in Figure 17, you will notice that there is a .got and .got.plt section within the data segment, and to enable full RELRO these are normally merged into one section but for our purposes that is not necessary since the tool I designed ‘relros’ marks both of them as read-only.

Overview of Statically Linked ELF

A high level overview can be seen with the ftrace tool, shown in Figure 18. Most of the heavy lifting that would normally take place in the dynamic linker is performed by the function generic_start_main() which in addition to other tasks also performs various relocations and fixups to all the many sections in the data segment, including the .got.plt section, in which case you can setup a few watch points to observe that early on there is a function that inquires about CPU information such as the CPU cache size, which allows glibc to intelligently determine which version of a given function, such as strcpy(), should be used.

In Figure 19, we set watch points on the GOT entries for several shared library routines and notice that generic_start_main() serves, in some sense, much like a dynamic linker. Its job is largely to perform relocations and fixups. So in both cases the GOT entry for a given libc function had its PLT stub address replaced with the most efficient version of the function given the CPU cache size looked up by certain glibc init code (i.e. __cache_sysconf()). Since this a somewhat high level overview I will not go into every function, but the important thing is to see that the PLT/GOT is updated with a libc function, and can be poisoned, especially since RELRO is not compatible with statically linked executables. This leads us into the solution, or possible solutions, including our very own experimental prototype named relros, which uses some ELF trickery to inject code that is called by a trampoline that has been placed in a very specific spot. It is necessary to wait until generic_start_main() has finished all of its writes to the memory areas that we intend to mark as read-only before we invoke our enable_relro() routine.

30 unzip pocorgtfo18.pdf checksec.sh # http://www.trapkit.de/tools/checksec.html
31 git clone https://github.com/elfmaster/ftrace

38
$ gcc -static -Wl,-z,relro,-z,now test.c -o test
$ readelf -l test
Elf file type is EXEC (Executable file)
Entry point 0x4008b0
There are 6 program headers, starting at offset 64

Program Headers:
<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>VirtAddr</th>
<th>PhysAddr</th>
<th>FileSiz</th>
<th>MemSiz</th>
<th>Flags</th>
<th>Align</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>R E</td>
<td>200000</td>
</tr>
<tr>
<td>LOAD</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>RW</td>
<td>200000</td>
</tr>
<tr>
<td>NOTE</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>R</td>
<td>4</td>
</tr>
<tr>
<td>TLS</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>R</td>
<td>8</td>
</tr>
<tr>
<td>GNU_STACK</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>RW</td>
<td>10</td>
</tr>
<tr>
<td>GNU_RELRO</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>R</td>
<td>1</td>
</tr>
</tbody>
</table>

Section to Segment mapping:
Segment Sections...
00 .note.ABI-tag .note.gnu.build-id .rela.plt .init .plt .text __libc_freeres_fn
    __libc_thread_freeres_fn .fini .rodata __libc_subfreeres __libc_atexit
    .stapd.base __libc_thread_subfreeres .eh_frame .gcc_except_table
01 .tdata .init_array .fini_array .jcr .data.rel.ro .got .got.plt .data .bss
    __libc_freeres_ptrs
02 .note.ABI-tag .note.gnu.build-id
03 .tdata .tbss
04
05 .tdata .init_array .fini_array .jcr .data.rel.ro .got

Figure 17. RELRO is Broken for Static Executables

$ ftrace test_binary
LOCAL_call@0x404fd0::__libc_start_main()
LOCAL_call@0x404fd0::get_common_indeces.constprop.1()
(RETURN VALUE) LOCAL_call@0x404fd0::get_common_indeces.constprop.1() = 3
LOCAL_call@0x404c0::generic_start_main() (RETURN VALUE) LOCAL_call@0x447cb0::dl_aux_init() = 7ffec5360bf9
LOCAL_call@0x449b0::dl_discover_osversion(0x7ffec5360be8)
LOCAL_call@0x465e0::uname() LOCAL_call@0x465e0::uname()
<truncated>

Figure 18. FTTracing a Static ELF
Figure 19. Exploring a Static ELF with GDB
A Second Implementation

My first prototype had to be written quickly due to time constraints. This current implementation uses an injection technique that marks the PT_NOTE program header as PT_LOAD, and we therefore create a second text segment effectively.

In the generic_start_main() function (Figure 20) there is a very specific place that we must patch and it requires exactly a five byte patch. (call <imm>.) As immediate calls do not work when transferring execution to a different segment, an lcall (far call) is needed which is considerably more than five bytes. The solution to this is to switch to a reverse text infection which will keep the enable_relro() code within the one and only code segment. Currently though we are being crude and patching the code that calls main().

Currently we are overwriting six bytes at 0x405b54 with a push enable_relro; ret set of instructions, shown in Figure 21. Our enable_relro() function mprotects the part of the data segment denoted by PT_RELRO as read-only, then calls main(), then sys_exits. This is flawed since none of the deinitialization routines get called. So what is the solution?

Like I mentioned earlier, we keep the enable_relro() code within the main programs text segment using a reverse text extension, or a text padding infection. We could then simply overwrite the five bytes at 0x405b46 with a call <offset> to enable_relro() and then that function would make sure we return the address of main() which would obviously be stored in  %rax. This is perfect since the next instruction is callq *%rax, which would call main() right after RELRO has been enabled, and no instructions are thrown out of alignment. So that is the ideal solution, although it doesn’t yet handle the problem of .tdata being at the beginning of the data segment, which is a problem for us since we can only use mprotect on memory areas that are multiples of a PAGE_SIZE.

A more sophisticated set of steps must be taken in order to get multi-threaded applications working with RELRO using binary instrumentation. Other solutions might use linker scripts to put the thread data and bss into their own data segment.

Notice how we patch the instruction bytes starting at 0x405b4f with a push %ret sequence, corrupting subsequent instructions. Nonetheless this is the prototype we are stuck with until I have time to make some changes.

So let’s take a look at this RelroS application. First we see that this is not a dynamically linked executable.

We observe that there is only a r+x text segment, and a r+w data segment, with a lack of read-only memory protections on the first part of the data segment.

We apply RelroS to the executable with a single command.

We observe that read-only relocations have been enforced by our patch that we instrumented into the binary called test.

Notice after we applied relros on ./test, it now has a 4096 area in the data segment that has been marked as read-only. This is what the dynamically linker accomplishes for dynamically linked executables.

---

32 Please note that it uses libelfmaster which is not officially released yet. The use of this library is minimal, but you will need to rewrite those portions if you intend to run the code.
33 unzip pocorgtfo18.pdf relros.c

41
Figure 20. Unpatched `generic_start_main()`.

Figure 21. Patched `generic_start_main()`.
So what are some other potential solutions for enabling RELRO on statically linked executables? Aside from my binary instrumentation project that will improve in the future, this might be fixed either by tricky linker scripts or by the glibc developers.

Write a linker script that places .tbss, .tdata, and .data in their own segment and the sections that you want readonly should be placed in another segment, these sections include .init_array, .fini_array, .jcr, .dynamic, .got, and .got.plt. Both of these PT_LOAD segments will be marked as PF_R|PF_W (read+write), and serve as two separate data segments. A program can then have a custom function—but not a constructor—that is called by main() before it even checks argc and argv. The reason we don’t want a constructor function is because it will attempt to mprotect readonly permissions on the second data segment before the glibc init code has finished performing its fixups which require write access. This is because the constructor routines stored in .init section are called before the write instructions to the .got, .got.plt sections, etc.

The glibc developers should probably add a function that is invoked by generic_start_main() right before main() is called. You will notice there is a _dl_protect_relo() function in statically linked executables that is never called.

ASLR Issues

ASLR requires that an executable is ET_DYN unless VMA mirroring is used for ET_EXEC ASLR. A statically linked executable can only be linked as an ET_EXEC type executable.

```
$ gcc -static -fPIC -pie test2.c -o test2
ld: x86_64-linux-gnu/5/crtbeginT.o: relocation R_X86_64_32 against `.TMC_END_' can not be used when making a shared object; recompile with -fPIC
x86_64-linux-gnu/5/crtbeginT.o: error adding symbols: Bad value
collect2: error: ld returned 1 exit status
```

ASLR requires that we need. So what to do?

This means that you can remove the -pie flag and end up with an executable that uses position independent code. But it does not have an address space layout that begins with base address 0, which is what we need. So what to do?

$ gcc -static -fPIC test2.c -o test2
$ ./static_to_dyn ./test2
$ gcc -static -fPIC test2.c -o test2
$ ./static_to_dyn ./test2
Setting e_entry to 8b0
$ ./test2
Segmentation fault (core dumped)

Alas, a quick look at the binary with objdump will prove that most of the code is not using IP relative addressing and is not truly PIC. The PIC version of the glibc init routines like _start lives in /usr/lib/X86_64-linux-gnu/Scrt1.o, so we may have to start thinking outside the box a bit about what a statically linked executable really is. That is, we might take the -static flag out of the equation and begin working from scratch!

Perhaps test2.c should have both a _start() and a main(), as shown in Figure 23. _start() should have no code in it and use __attribute__((weak)) so that the _start() routine in Scrt1.o can override it. Or we can compile

A Hybrid Solution

The linker may not be able to perform this task yet, but I believe we can. A potential solution exists in the idea that we can at least compile a statically linked executable so that it uses position independent code (IP relative), although it will still maintain an absolute address space. So here is the algorithm as follows from a binary instrumentation standpoint.

First we’ll compile the executable with -static -fPIC, then static_to_dyn.c adjusts the executable. First it changes the ehdr->e_type from ET_EXEC to ET_DYN. It then modifies the phdrs for each PT_LOAD segment, setting phdr[TEXT].p_vaddr and .p_offset to zero, phdr[DATA].p_vaddr to 0x2000000 + phdr[DATA].p_offset. It sets ehdr->e_entry to ehdr->e_entry - old_base. Finally, it updates each section header to reflect the new address range, so that GDB and objdump can work with the binary.

ASLR Solutions

I haven’t personally spent enough time with the linker to see if it can be tweaked to link a static executable that comes out as an ET_DYN object, which should also not have a PT_INTERP segment since it is not dynamically linked. A quick peak in src/linux/fs/binfmt_elf.c, shown in Figure 22, will show that the executable type must be ET_DYN.
} else if (loc->elf_x.e_type == ET_DYN) {
    /* Try and get dynamic programs out of the way of the
     * default mmap base, as well as whatever program they
     * might try to exec. This is because the brk will
     * follow the loader, and is not movable. */
    load_bias = ELF_ET_DYN_BASE - vaddr;
    if (current->flags & PF_RANDOMIZE)
        load_bias += arch_mmap_rnd();

    if (! load_addr_set) {
        load_addr_set = 1;
        load_addr = (elf_ppnt->p_vaddr - elf_ppnt->p_offset);
        if (loc->elf_x.e_type == ET_DYN) {
            load_bias += error -
                ELF_PAGESTART(load_bias + vaddr);
            load_addr += load_bias;
            reloc_func_desc = load_bias;
        }
    }
}

Figure 22. src/linux/fs/binfmt_elf.c

Diet Libc\textsuperscript{34} with IP relative addressing, using it instead of glibc for simplicity. There are multiple possibilities, but the primary idea is to start thinking outside of the box. So for the sake of a PoC here is a program that simply does nothing but check if \texttt{argc} is larger than one and then increments a variable in a loop every other iteration. We will demonstrate how ASLR works on it. It uses _start() as its \texttt{main()}, and the compiler options will be shown below.

\begin{figure}[h]
\begin{Verbatim}
$ gcc -nostdlib -fPIC test2.c -o test2
$ ./test2 arg1
$ pmap 'pidof test2'
17370: ./test2 arg1
0000000000400000 4K r-x— test2
000000000601000 4K rw— test2
00007ffceeda000 132K rw— | stack |
00007ffced20000 8K r— | anon |
00007ffced22000 8K r-x— | anon |
fffffff600000 4K r-x— | anon |
total 160K
\end{Verbatim}
\end{figure}

\begin{figure}[h]
\begin{Verbatim}
$ ./static_to_dyn test2
$ ./test2 arg1
$ pmap 'pidof test2'
17622: ./test2 arg1
0000565271e41000 4K r-x— test2
0000565272d20000 4K rw— test2
00007ffceeda000 132K rw— | stack |
00007ffceedf0000 8K r— | anon |
00007ffceedf0000 8K r-x— | anon |
fffffff600000 4K r-x— | anon |
total 160K
\end{Verbatim}
\end{figure}

Now notice that the text and data segments for \texttt{test2} are mapped to a random address space. Now we are talking! The rest of the homework should be fairly straightforward. Extrapolate upon this work and find more creative solutions until the GNU folks have the time to address the issues with some more elegance than what we can do using trickery and instrumentation.

\begin{figure}[h]
\begin{Verbatim}
$ unzip pocorgtfo18.pdf dietlibc.tar.bz2
\end{Verbatim}
\end{figure}

ASLR is not present, and the address space is just as expected on a 64 class ELF binary in Linux. So let’s run \texttt{static_to_dyn.c} on it, and then try again.

\textsuperscript{34}unzip pocorgtfo18.pdf dietlibc.tar.bz2
Figure 23. First Draft of test2.c

Improving Static Linking Techniques

Since we are compiling statically by simply cutting glibc out of the equation with the -nostdlib compiler flag, we must consider that things we take for granted, such as TLS and system call wrappers, must be manually coded and linked. One potential solution I mentioned earlier is to compile dietlibc with IP relative addressing mode, and simply link your code to it with -nostdlib. Figure 24 is an updated version of test2.c which prints the command line arguments.

Now we are actually building a statically linked binary that can get command line args, and call statically linked in functions from Diet Libc.

$ gcc -nostdlib -c -fPIC test2.c -o test2.o
$ gcc -nostdlib test2.o \n /usr/lib/diet/lib-x86_64/libc.a -o test2
$ ./test2 arg1 arg2

Now we can run static_to_dyn from Figure 25 to enforce ASLR. The first two sections are happily randomized!

$ ./static_to_dyn test2
$ ./test2 arg1 arg2
$ pmap `pidof test`
24411: ./test2 arg1 arg2
0000564cf542f000 8K r-x--- test2
0000564cf5651000 4K rw--- test2
0000564cf5651000 132K rw--- [ stack ]
0000564cf5651000 8K r--- [ anon ]
0000564cf5657000 8K r-x--- [ anon ]
0000564cf5660000 4K r-x--- [ anon ]
total 164K

Note that first I downloaded the dietlibc source code and edited the Makefile to use the -fPIC flag which will enforce IP-relative addressing within dietlibc.

unzip pocorgtfo18.pdf static_to_dyn.c
#include <stdio.h>

/* Make sure we have a data segment for testing purposes */
static int test_dummy = 5;

int _start() {
   int argc;
   long *args;
   long *rbp;
   int i;
   int j = 0;

   /* Extract argc from stack */
   asm __volatile__("mov 8(%%rbp), %%rcx " : "=c" (argc));
   /* Extract argv from stack */
   asm __volatile__("lea 16(%%rbp), %%rcx " : "=c" (args));
   for (i = 0; i < argc; i++) {
      sleep(10); /* long enough for us to verify ASLR */
      printf("%s\n", args[i]);
   }
   exit(0);
}

Figure 24. Updated test2.c.

Summary

In this paper we have cleared some misconceptions surrounding the attack surface of a statically linked executable, and which security mitigations are lacking by default. PLT/GOT attacks do exist against statically linked ELF executables, but RELRO and ASLR defenses do not.

We presented a prototype tool for enabling full RELRO on statically linked executables. We also engaged in some work to create a hybridized approach between linking techniques with instrumentation, and together were able to propose a solution for making static binaries that work with ASLR. Our solution for ASLR is to first build the binary statically, without glibc.
```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <elf.h>
#include <search.h>
#include <sys/types.h>
#include <sys/time.h>
#include <fcntl.h>
#include <link.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define HUGE_PAGE 0x200000

int main(int argc, char **argv) {
    ElfW(Ehdr) *ehdr;
    ElfW(Phdr) *phdr;
    ElfW(Shdr) *shdr;
    uint8_t *mem;
    int fd;
    struct stat st;
    uint64_t old_base; /* original text base */
    uint64_t new_data_base; /* new data base */
    char *StringTable;

    fd = open(argv[1], O_RDWR);
    if (fd < 0) {
        perror("open");
        goto fail;
    }
    fstat(fd, &st);
    mem = mmap(NULL, st.st_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (mem == MAP_FAILED) {
        perror("mmap");
        goto fail;
    }
    ehdr = (ElfW(Ehdr) *)mem;
    shdr = (ElfW(Shdr) *)&mem[ehdr->e_shoff];
    StringTable = (char *)&mem[shdr[ehdr->e_shstrndx].sh_offset];
    printf("Marking e_type to ET_DYN\n");
    ehdr->e_type = ET_DYN;
    printf(" Updating PT_LOAD segments to become relocatable from base 0\n");
    for (i = 0; i < ehdr->e_phnum; i++) {
        if (phdr[i].p_type == PT_LOAD && phdr[i].p_offset == 0) {
            old_base = phdr[i].p_vaddr;
            phdr[i].p_vaddr = 0UL;
            phdr[i].p_paddr = 0UL;
            phdr[i+1].p_vaddr = HUGE_PAGE + phdr[i+1].p_offset;
            phdr[i+1].p_paddr = HUGE_PAGE + phdr[i+1].p_offset;
        } else if (phdr[i].p_type == PT_NOTE) {
            phdr[i].p_vaddr = phdr[i].p_offset;
            phdr[i].p_paddr = phdr[i].p_offset;
        } else if (phdr[i].p_type == PT_TLS) {
            phdr[i].p_vaddr = HUGE_PAGE + phdr[i].p_offset;
            phdr[i].p_paddr = HUGE_PAGE + phdr[i].p_offset;
        } else if (phdr[i].p_type == PT_LOAD) {
            phdr[i].p_vaddr = phdr[i].p_offset;
            phdr[i].p_paddr = phdr[i].p_offset;
        } else if (phdr[i].p_type == PT_DYNAMIC) {
            phdr[i].p_vaddr = phdr[i].p_offset;
            phdr[i].p_paddr = phdr[i].p_offset;
        } else {
            printf("Setting %s sh_addr to %lx\n", &StringTable[shdr[i].sh_name], shdr[i].sh_addr);
        }
    }
    for (i = 0; i < ehdr->e_shnum; i++) {
        if (! (shdr[i].sh_flags & SHF_ALLOC)) {
            shdr[i].sh_addr = (shdr[i].sh_addr < old_base + HUGE_PAGE)
                ? 0UL : shdr[i].sh_addr;
            update_data_base = shdr[i].sh_offset;
            printf("Setting %lx\n", &StringTable[shdr[i].sh_name], shdr[i].sh_addr);
        }
    }
}
```