ASLR Protection for Statically Linked Executables

Jun 27

Written By Ryan O'Neill

Ryan “ElfMaster” O’Neill
Email: ryan.oneill@leviathansecurity.com
Twitter: @ryan_elfmaster

Target Audience: Software vendors, OS maintainers, Software engineers, & Security researchers.

1 Introduction

This paper provides insights into the more obscure security weaknesses of statically linked executables, including, but not limited to, the following:

- the glibc initialization code for statically linked executables
- what the attack surface looks like for statically linked executables
- why mitigations such as RELRO and ASLR are as important for statically linked executables as they are for dynamically linked executables
- common misunderstandings about RELRO, ASLR, and static executables: that static linking disables important security mitigations, leaving the program vulnerable

Currently, read-only relocations (RELRO) is a security mitigation (discussed in section 2.3) that is vitally important to statically linked executables. Unbeknownst to even the rare ELF binary gurus, this RELRO mitigation is absent and is falsely believed to be unimportant for static-linked executables. As we will demonstrate with some RE examples, statically and dynamically linked executables have the same attack surfaces. A second issue is that ASLR cannot be applied to a static executable, but like RELRO, is now a vital security mitigation. In this post, we explore in depth both the details of using RELRO and ASLR with static executables, and the solutions to each problem, followed by a POC for readers to extrapolate from or use to mitigate static executable security risks until a more complete solution is developed.

Static binaries are commonly used in common off-the-shelf software (COTS) for several reasons, such as avoiding dependence on dynamic library version compatibility problems. This ignores the security consequences of statically linked executables, in part due to certain scripts, such as checksec.sh[3], incorrectly reporting that statically linked executables having partial RELRO enabled.

The purpose of this treatise is to contribute to the security of the software developer life-cycle that should be applied to the growing number of OSs (operating systems) that are supporting the ELF binary format. Static executables are more of a prime target now than ever, as we will see in section 3.1, and RELRO and ASLR make exploitation far more difficult to achieve. If a piece of software has a memory corruption vulnerability, having the appropriate binary mitigations turned on has the potential to completely prevent an exploit from working. For example, in a scenario where the attacker has only a single pointer width write primitive, and there is no mprotect@PLT to mark the RELRO segment as writeable, RELRO would make the vulnerability unexploitable. As mentioned, we will explore this in more detail in section 3.1.

2 The State of Standard ELF Security Mitigations

Over the years, substantial improvements have been incorporated into glibc (GNU C Library), the GNU linker, and the dynamic linker in order to make various security mitigations possible, including full ASLR, which requires modifications to the compiler and linker. Pipacs, a well-respected and prolific security researcher known for PaX, has created many userland and kernel mitigation technologies ranging from ASLR to PAGEEXEC [4].

One of Pipacs' discoveries is that in addition to randomizing the address space of the heap, stack, and shared, one can also randomize the address space of the ELF executable itself, thus making ret2plt and ROP attacks in general more difficult. Two solutions were developed: first RandExec [5], and then a much more elegant solution which lead to PIE (position independent executable) binaries.

The idea behind PIE binaries is that you can create an ELF executable to have the same attributes as a shared library object: Position-independent code (PIC), and a base address of 0x0 so that the binary can be relocated by the kernel at runtime. The only differences between a regular shared library and a PIE executable are that the initialization code, and that executables must have a PT_INTERP segment to describe the path to the dynamic linker. A regular executable has the ELF file type ET_EXEC, whereas a PIE executable has a file type of ET_DYN, as do shared libraries. PIE executables use IP relative addressing mode to avoid hard-coding references to absolute addresses. A program that is an ELF ET_DYN and has a base address of 0x0 can be randomly relocated to a different base address every time it is run.

2.1 Related Work Supporting Static PIE

After writing the bulk of this paper, I discovered that there have been some attempts at getting static PIE executables into the mainline. However, from what I can tell, there is still not a standard option. I also discovered that there is a patch to add the static PIE option. For further information, see [6] and [7], two forum posts I found after some digging.

2.2 When Is Full ASLR Important?

When an executable runs privileged, such as sshd, it would ideally be compiled and linked into a PIE executable that allows for runtime relocation to a random address space, thus

hardening the attack surface into a far more hostile playing ground. Sensitive programs running as root should never be built as statically linked, and should almost always have all of the available binary mitigations turned on.

One reason not to use PIE binaries is that IP relative addressing can affect the program performance on various levels, as described in this interesting paper from a Red Hat team[8]. Occasionally, edge cases will arise in which one must turn off mitigations such as -fstack-protector in order to enable custom mitigations. But in general, everything from canaries to ASLR and full RELRO should be enabled for sensitive software whenever possible. For instance, sshd is nearly always built with all mitigations enabled, including full ASLR, meaning that sshd was built as a PIE executable. Try running `readelf -e /usr/sbin/sshd | grep DYN`, and you will see that the sshd is (most likely) built this way, although there are some exceptions, depending on the architecture.

2.3 A Primer on RELRO

Let's traverse over to another security mitigation that is less well known. RELRO is a security mitigation technique that has two modes (partial and full). By default, only the partial RELRO is enforced because it uses lazy linking, whereas full RELRO requires strict linking [1]. Strict linking has less efficient program loading time than lazy linking due to the dynamic linker binding/relocating immediately in strict linking rather than on-demand. But full RELRO can be very powerful for hardening the attack surface by marking specific areas in the data segment as read-only, secifically 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 function pointers into destructor routines, respectively.

3 A Holistic View of Static Linked Executables and Security

Developers are often use 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 requires many dynamic library dependencies, sometimes thousands of them. As a professional researcher in this area I've been aware of the more obvious pros and cons of statically linked ELF executables, but I was remiss to think that they don't suffer from the same ELF security problems as dynamically linked executables. In fact, to my surprise, I found that a statically linked executable is vulnerable to many of the same attacks as a dynamically linked executable, including those detailed here in the section 3.1:

3.1 Attack Points Protected by RELRO

- shared library injection for malware purposes
- Dtors (.fin_array) poisoning (note: only relevant in some statically linked executables)
- Got.plt poisoning (i.e., GOTPLT hijacking)

3.2 Why Under the RADAR So Long?
These vulnerabilities in statically linked executables went under the radar for so long because the .got.plt section was not always used as a clever optimization. It simply did not exist and therefore presented no attack surface. I do not know the exact date when this .got.plt optimization was introduced.

3.3 Full RELRO Protection vs. Partial

Full RELRO protects all sections (.got.plt, .got, .init_array, .fini_array, .dynamic, and .jcr) while partial RELRO omits protecting .got.plt because it needs to be updated throughout the life of the process to support on-demand dynamic linking. This is a problem, because it leaves the .got.plt exposed as an attack surface.

Static executables by default have no RELRO enabled, and yet still leave the .got.plt attack surface exposed, even though static executables only update the .got.plt during process initialization time. If an attacker can find a .got.plt section, they can corrupt critical function pointers. So, what is the solution for protecting static executables?

Note: .init_array and .fini_array are the new name for .ctors and .dtors, and have corresponding SHT_INIT_ARRAY and SHT_FINI_ARRAY types.

3.4 Deeper Into the Attack Surface
Discovering that the .got.plt is an exposed attack surface in statically linked binaries surprised me, as well as several adept ELF researchers that I know. Let's look all the ways in which RELRO can protect an executable.

3.4.1 Shared Library Injection Protection

Although shared library injection protection was not the original purpose of RELRO, it can be combined with various implementations of DEP, such as PaX mprotect() restrictions, and prevent runtime malware attacks. For example, shared library function redirection is foiled by the fact that PaX disallows PTRACE_POKETEXT to read-only segments.

3.4.2 The Same Old Exploitation Techniques Apply

From an exploitation standpoint, things becomes more interesting when you realize that the .got.plt section is still a relevant attack surface in statically linked executables. We will discuss the .got.plt's purpose shortly, but for now, it is important to note that the .got.plt contains function pointers to libc routines and has been historically exploited in dynamically linked executables. The .init_array and .fini_array function pointers respectively point to initialization and destructor routines.

Specifically, .fini_array, also known as .dtors, has been leveraged to achieve code execution in many types of exploits, although its abuse are likely not as ubiquitous as the .got.plt section. In the following sections, we analyze the lack of security mitigations in statically

linked executables, and in doing so, we will explore some reverse engineering and binary mitigation techniques. This analysis will demonstrate to the reader that the attack surface for statically linked executables is virtually the same as that of a dynamically linked executable. This information is important since binary mitigations such as RELRO and ASLR do not work with statically linked executables with the current tool chain. If you are bored, just wait. Shortly we will get into the nuts and bolts of things and explore some innovative hacks for applying ASLR and RELRO to statically linked executables.

3.4.3 RELRO Ambiguities In Statically Linked Executables

The following static binary was built with full RELRO enabled using the 'gcc -static -Wl,-z,relro,-z,now' command, after which even the savvy reverser might be fooled into thinking that RELRO is enabled. Partial RELRO and full RELRO are both incompatible with statically linked executables at this point, because the dynamic linker is responsible for re-mapping and mprotecting the common attack points within the data segment, such as the .got.plt, and as shown in the output below there is no program header of type PT_INTERP to specify an interpreter, nor would we expect to see one in a statically linked executable since they don't use dynamic linking.

The default linker script is what directs the linker to create the GNU_RELRO segment, even though this segment serves no current purpose. We designed a solution for enabling RELRO on statically linked executables, which makes use of the PT_GNU_RELRO program header. The PT_GNU_RELRO program header shares the same p_vaddr and p_offset as the data segment's program header, since this is where the .init_array, .fini_array, .jcr, .dynamic, .got, and .got.plt sections are going to live in memory. The size of the PT_GNU_RELRO in memory is described by the program header's p_memsz field, which should be aligned up to the next PAGE_SIZE in order to acquire the 'len' value that the dynamic linker would pass to mprotect() in order to mark the page(s) read-only. Let's take a closer look at the program headers and the sections that map to them.

Here are the program headers of a statically linked ELF executable:

```
$ readelf -l test

Elf file type is EXEC (Executable file)
Entry point 0x4008b0
```

There are 6 program headers, starting at offset 64:

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000cbf67 0x00000000000cbf67  R E    200000
  LOAD           0x00000000000cceb8 0x00000000006cceb8 0x00000000006cceb8
                 0x0000000000001cb8 0x0000000000003570  RW     200000
  NOTE           0x0000000000000190 0x0000000000400190 0x0000000000400190
                 0x0000000000000044 0x0000000000000044  R      4
  TLS            0x00000000000cceb8 0x00000000006cceb8 0x00000000006cceb8
                 0x0000000000000020 0x0000000000000050  R      8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x00000000000cceb8 0x00000000006cceb8 0x00000000006cceb8
                 0x0000000000000148 0x0000000000000148  R      1
```

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 .stapsdt.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
```

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. Typically, no more than the first 4096 bytes of the data segment need protection. The static executable, however, was not created with security in mind, despite the GNU_RELRO segment. One telltale sign of this is that the .tdata section, which contains TLS data, is the first section in the data segment and should it become marked PROT_READ with mprotect(), then multi-threaded programs would fail to run correctly. If the glibc ld/gcc developers had had RELRO in mind, they would have likely put the .tdata section in between the .data section and the .bss section.

3.4.4 An Important Side-note

As a @ulexec pointed out, the .init_array/.fini_array sections don't work in glibc statically linked executables. The .got.plt, on the other hand, is very active since it is used as an optimization for libc routines in static executables. The .got.plt is the largest threat, and it is used in glibc's model of static linking, whereas what we would traditionally call the .ctors and .dtors do not get used, although they exist, usually as .init_array and .fini_array. However, a static executable is not always as rigid as using the -static flag, as will be demonstrated when a static ELF is removed from the context of glibc's rigid linker script(s) and standard init/deinit routines.

Because -static is not as rigid as it seems, we must consider all ELF sections (not just the .got.plt) that are typically protected by RELRO as virtually identical in activity and behavior within statically linked executables, until proven otherwise on a case by case basis. We know that the .got.plt serves as an optimization for libc routines, as we will demonstrate shortly, and this .got.plt table of function pointers is an excellent attack surface and has been used in the wild for dynamically linked executables since the early 2000s, as shown in Nergal's paper published many years ago.[9]

3.4.5 checksec.sh Fails to Provide Accurate Information

checksec.sh [3] uses the GNU_RELRO segment as one of the markers to denote whether or not RELRO is enabled on a binary. In the case of statically compiled binaries, checksec.sh 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. To make this more concrete, let's take a lightweight tour through the glibc init code of a statically linked executable.

In the above output, there is a .got and .got.plt section within the data segment. Usually, enabling full RELRO requires that these are merged into one single '.got' section. However, we designed a tool, 'RelroS', that does not require merging, and instead marks them both as read-only.

4 A Deeper Overview of Static Linking and Attack Surfaces

NOTE: A high-level overview can be seen with the ftrace tool https://github.com/elfmaster/ftrace, which performs function-level tracing.

```
$ ftrace test_binary
LOCAL_call@0x404fd0:__libc_start_main()
LOCAL_call@0x404f60:get_common_indeces.constprop.1()
(RETURN VALUE) LOCAL_call@0x404f60: get_common_indeces.constprop.1() = 3
LOCAL_call@0x404cc0:generic_start_main()
LOCAL_call@0x447cb0:_dl_aux_init() (RETURN VALUE) LOCAL_call@0x447cb0:
_dl_aux_init() = 7ffec5360bf9
LOCAL_call@0x4490b0:_dl_discover_osversion(0x7ffec5360be8)
LOCAL_call@0x46f5e0:uname() LOCAL_call@0x46f5e0:__uname()
<truncated>
```

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. This allows one the ability to set up a few watch points to observe that early on there is a function that inquires about CPU information, such as the CPU cache size. This function allows the glibc init code to intelligently determine which version of a given function, such as strcpy(), should be used for optimizations.

Notice that when we set watch points on the GOT entries for several shared library routines, the generic_start_main() serves in one a sense, as a similar mechanism to the dynamic linker, as its job is largely to perform relocations and fixups.

-- Inside an exploratory GDB session with a static binary --

```
(gdb) x/gx 0x6d0018 /* .got.plt entry for strcpy */
0x6d0018:    0x000000000043f600
(gdb) watch *0x6d0018
Hardware watchpoint 3: *0x6d0018
(gdb) x/gx       /* .got.plt entry for memmove */
0x6d0020:    0x0000000000436da0
(gdb) watch *0x6d0020
Hardware watchpoint 4: *0x6d0020
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/elfmaster/git/libelfmaster/examples/static_binary
Hardware watchpoint 4: *0x6d0020
Old value = 4195078
New value = 4418976
0x0000000000404dd3 in generic_start_main ()
(gdb) x/i 0x436da0
   0x436da0 <__memmove_avx_unaligned>:  mov    %rdi,%rax
(gdb) c
```

Continuing.

```
Hardware watchpoint 3: 0x6d0018
*Old value = 4195062
New value = 4453888
0x0000000000404dd3 in generic_start_main ()
(gdb) x/i 0x43f600
   0x43f600 <__strcpy_sse2_unaligned>:  mov    %rsi,%rcx
(gdb)
```

4.1 .got.plt Inspection with GDB

In both cases above, 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, for example, __cache_sysconf(). Since this a somewhat high-level overview, I will not go into every function, but the important thing is that the .got.plt is updated with a libc function, and can be poisoned, especially since RELRO is not compatible with statically linked executables.

This leads us to several possible solutions, including our experimental prototype, RelroS (read-only relocations for static ELF), 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.

4.2 RelroS (Readonly-Relocations for Static ELF) A Solution

The initial (and only) version of RelroS was written quickly due to time constraints. Consequently, there are several problems in the current implementation, but I explain below how to resolve them. 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. Furthermore, in the generic_start_main() function, there is a very specific place that we must patch, and it requires exactly a 5 byte patch (i.e., call <imm>).

Unfortunately, immediate calls do not work when transferring execution to a different segment. Instead, an lcall (far call) is needed, which is considerably more than 5 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()

```
  405b46:       48 8b 74 24 10          mov    0x10(%rsp),%rsi
  405b4b:       8b 7c 24 0c             mov    0xc(%rsp),%edi
  405b4f:       48 8b 44 24 18          mov    0x18(%rsp),%rax /* store main() addr
*/
  405b54:       ff d0                   callq  *%rax /* call main() */
  405b56:       89 c7                   mov    %eax,%edi
  405b58:       e8 b3 de 00 00          callq  413a10 <exit>
```

In the current example, we overwrite 6 bytes at 0x405b54 with a 'push $enable_relro; ret' set of instructions. Our enable_relro() function mprotects the part of the data segment denoted by PT_RELRO as read-only, then calls main(), then sys_exit's. This is flawed since none of the code after main(), including deinitialization routines, get called.

The solution is to keep the enable_relro() code within the main programs text segment using a reverse text extension, or a text padding infection. This allows us to simply overwrite the 5 bytes at 0x405b46 with a 'call <offset>' to enable_relro() and that function would ensure we return the address of main() which would be stored in %rax. Since the next instruction is 'callq *%rax', which would call main() right after RELRO has been enabled, no instructions are thrown out of alignment.

Thus far, this solution is ideal. However, .tdata being at the beginning of the data segment is a problem since we can only use mprotect() on memory areas that are multiples of a PAGE_SIZE. Thus, a slightly more sophisticated set of steps must be taken to get multi-threaded applications working with RELRO using binary instrumentation (Alternatively, we could solve the problem by using linker scripts to put the thread data and bss into its own data segment).

In the current prototype, we patch the instruction bytes starting at 0x405b4f with a push/ret sequence, corrupting the following instructions. This is a temporary fix that needs to be addressed in future prototypes.

```
  405b46:       48 8b 74 24 10          mov    0x10(%rsp),%rsi
  405b4b:       8b 7c 24 0c             mov    0xc(%rsp),%edi
  405b4f:       48 8b 44 24 18          mov    0x18(%rsp),%rax
  405b54:       68 f4 c6 0f 0c          pushq  $0xc0fc6f4
  405b59:       c3                      retq
  /*
   * The following bad instructions are never crashed on because
   * the previous instruction returns into enable_relro() which calls
   * main() on behalf of this function, and then sys_exit's out.
   */
  405b5a:       de 00                   fiadd  (%rax)
  405b5c:       00 39                   add    %bh,(%rcx)
  405b5e:       c2 0f 86                retq   $0x860f
  405b61:       fb                      sti
  405b62:       fe                      (bad)
  405b63:       ff                      (bad)
  405b64:       ff                      (bad)
```

We see that this is not a dynamically linked executable

```
$ readelf -d test
There is no dynamic section in this file.
```

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.

```
$ ./test &
[1] 27891
$ cat /proc/`pidof test`/maps
00400000-004cc000 r-xp 00000000 fd:01 4856460 /home/elfmaster/test
006cc000-006cf000 rw-p 000cc000 fd:01 4856460 /home/elfmaster/test
<truncated>
```

We apply RelroS to the executable with a single command

```
$ ./relros ./test
injection size: 464
main(): 0x400b23
```

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

```
$ ./test &
[1] 28052
$ cat /proc/`pidof test`/maps
00400000-004cc000 r-xp 00000000 fd:01 10486089 /home/elfmaster/test
006cc000-006cd000 r--p 000cc000 fd:01 10486089 /home/elfmaster/test
006cd000-006cf000 rw-p 000cd000 fd:01 10486089 /home/elfmaster/test
<truncated>
$
```

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

Currently, we are working to improve our binary instrumentation project [11] for enabling RELRO on statically linked executables. In sections 4.3 and 4.4, I discuss two other potential solutions to this problem.

4.3 Linker Scripts and Custom Function

One possible method for enabling RELRO on static executables is to write a linker script that separates .tbss, .tdata, and .data into their own segment and then place the sections that should be read-only (i.e., .init_array, .fini_array, .jcr, .dynamic, .got, and .got.plt) in another segment.. This allows each PT_LOAD segment to be individually marked as PF_R|PF_W (read+write) so that they serve as two separate data segments.

The separate segments allow a program to have a custom function (not a constructor) that is called by main() before it even checks argc/argv. A custom rather than a constructor function should be used because the constructor routines stored in .init section are called before the write instructions to the .got, .got.plt sections, and so forth. A constructor function would attempt to mprotect() read-only permissions on the second data segment before the glibc init code has finished performing its fixups, which require write access, and thus fail to function.

4.4 GLIBC Developers Could Fix It

Another solution to enabling RELRO on static executables is for glibc developers to add a function that is invoked by generic_start_main() right before main() is called. At present, there is a _dl_protect_relro() function in statically linked executables that is never called.

5 ASLR Issues

As mentioned before, binary mitigations such as ASLR cannot be applied to static executables with the current tool chain. ASLR requires that an executable is ET_DYN unless RANDEXEC [5] 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
/usr/bin/ld: /usr/lib/gcc/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
/usr/lib/gcc/x86_64-linux-gnu/5/crtbeginT.o: error adding symbols: Bad value
collect2: error: ld returned 1 exit status
```

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

5.1 ASLR Solutions

I haven't personally spent enough time with the glibc linker to see if it can be tweaked to link a static executable that comes out as an ET_DYN object. It is worth nothing that such an executable should not have a PT_INTERP segment since it is not dynamically linked. Due to my own time constraints I'd like to leave this as an exercise for the reader, and maybe there are some solutions that I'm not aware of.

The following code from the ELF kernel loader will further concretize the fact that the executable type must be ET_DYN in order for it to be relocated to a randomly generated address.

A quick peek in src/linux/fs/binfmt_elf.c reveals this code at line 916

```
 line 916:      } else if (loc->elf_ex.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();
```

... THEN ...

```
line 941:     if (!load_addr_set) {
               load_addr_set = 1;
               load_addr = (elf_ppnt->p_vaddr - elf_ppnt->p_offset);
               if (loc->elf_ex.e_type == ET_DYN) {
                     load_bias += error -
                         ELF_PAGESTART(load_bias + vaddr);
                     load_addr += load_bias;
                     reloc_func_desc = load_bias;
               }
       }
```

5.2 ASLR Binary Instrumentation/Linker Hybrid Solutions

The linker may not be able to perform this task yet, but I believe 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). The following is the algorithm as follows from a binary instrumentation standpoint:

1. gcc -static -fPIC test2.c -o test2 --static_to_dyn.c uses this algorithm

2. Modify ehdr->e_type from ET_EXEC to ET_DYN

3. Modify the phdr's for each PT_LOAD segment (text and data segment respectively)

a. phdr[TEXT].p_vaddr = 0x00000000;

b. phdr[TEXT].p_offset = 0x00000000;

c. phdr[DATA].p_vaddr = 0x200000 + phdr[DATA].p_offset;

4. ehdr->e_entry = ehdr->e_entry - old_base;

5. Update each section header to reflect new address range of program headers.

otherwise GDB and objdump won't work with the binary:

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

5.3 Expanding Our Perspectives on Statically Linked Executables for the Sake of ASLR
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. I believe we may have to start with a novel approach such as taking the '-static' gcc option out of the equation and begin working from scratch. Following are several starting points for a solution.

Perhaps test2.c should have both a _start() and a main(), and _start() should have no code in it and use the __attribute__((weak)) so that the _start() routine in Scrt1.o can override it. Another possible solution would be to compile diet libc with IP relative addressing and use it instead of glibc for more simplicity. There are multiple possibilities, but the primary idea is to start thinking outside of the box. For the sake of a POC, here is a program that simply does nothing but check if argc is larger than 1 and then increments a variable in a loop every other iteration. We will demonstrate how ASLR works on it. It uses _start() as its main(), and the compiler options will be shown below.

```
*** PoC of simple static binary made to ASLR ***

/* 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));

      if (argc > 2) {
             for (i = 0; i < 100000000000; i++)
                   if (i % 2 == 0)
                         j++;
      }
      return 0;
}

$ gcc -nostdlib -fPIC test2.c -o test2
$ ./test2 arg1

$ pmap `pidof test2`
17370:   ./test2 arg1
0000000000400000      4K r-x-- test2
0000000000601000      4K rw--- test2
00007ffcefcca000    132K rw---   [ stack ]
00007ffcefd20000      8K r----   [ anon ]
00007ffcefd22000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total              160K
$
```

Notice that ASLR is not present, and the address space is just as expected on a 64 class ELF binary in Linux. Let's run our static_to_dyn.c program on it, and then try again.

```
$ ./static_to_dyn test2
$ ./test2 arg1
$ pmap `pidof test2`
17622:   ./test2 arg1
0000565271e41000      4K r-x-- test2
0000565272042000      4K rw--- test2
00007ffc28fda000    132K rw---   [ stack ]
00007ffc28ffc000      8K r----   [ anon ]
00007ffc28ffe000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total              160K
```

Notice that the text and data segment for test2 are mapped in at 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.

5.4 Improving our 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. Here is our updated test2.c code which prints the command line arguments:

```
*** updated test2.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);
}
```

As a side note, the reader can figure out how to get char **envp. I left it as an exercise. Now we are actually building a statically linked binary that can get command line args and call statically linked functions from diet libc:

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

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

Now we can run our static_to_dyn tool on it to enforce ASLR:

```
$ ./static_to_dyn test2
$ ./test2 foo bar
$ pmap `pidof test`
24411:   ./test2 foo bar
0000564cf542f000      8K r-x-- test2 # Notice ASLR!
0000564cf5631000      4K rw--- test2 # Notice ASLR!
00007ffe98c8e000    132K rw---   [ stack ]
00007ffe98d55000      8K r----   [ anon ]
00007ffe98d57000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total              164K
```

*** static_to_dyn.c ***

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <elf.h>
#include <sys/types.h>
#include <search.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;
      int i;
      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;
	phdr = (ElfW(Phdr) *)&mem[ehdr->e_phoff];
	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;
			new_data_base = phdr[i].p_vaddr;
		}
	}
	/*
	 * If we don't update the section headers to reflect the new address
	 * space then GDB and objdump will be broken with this binary.
	 */
	for (i = 0; i < ehdr->e_shnum; i++) {
		if (!(shdr[i].sh_flags & SHF_ALLOC))
			continue;
		shdr[i].sh_addr = (shdr[i].sh_addr < old_base + HUGE_PAGE) ?
		    0UL + shdr[i].sh_offset : new_data_base + shdr[i].sh_offset;
		printf("Setting %s sh_addr to %#lx\n", &StringTable[shdr[i].sh_name],
		    shdr[i].sh_addr);
	}
	printf("Setting new entry point: %#lx\n", ehdr->e_entry - old_base);
	ehdr->e_entry = ehdr->e_entry - old_base;
	munmap(mem, st.st_size);
	exit(0);
	fail:
		exit(-1);
}
```

6 Summary

The purpose of this paper is to clear up misconceptions about – and help to demystify the ambiguity surrounding – what the attack surface is within a statically linked executable, and what security mitigations are lacking by default. RELRO and ASLR will not work with statically linked executables. However, in this paper we presented the "RelroS" tool, which is a prototype for enabling full RELRO on statically linked executables. We also created a hybridized approach combining compiling/linking techniques with instrumentation

techniques, and together with our RELRO enablement, were able to propose a solution for making static binaries that work with ASLR. Currently, our solution for RELRO will only work on traditionally built static binaries (e.g., -static flag) since the tool patches a glibc initialization function. Our solution for ASLR is to first build the binary statically but without glibc.

6.1 Homework for the reader

Currently relros.c and static_to_dyn.c can be applied individually but not simultaneously; this is because static_to_dyn.c does not work on standard statically linked executables, and relros.c works only on standard static linked executables. Ideally, we want a tool that can apply ASLRand RELRO on the same statically linked executable. Some general steps to accomplishing this:

1. Create a static binary using the approach in section 5.4. This combines the -nostdlib flag with a version of dietlibc that's been compiled with position independent code.
2. Use the existing static_to_dyn.c source code to convert the binary into an ET_DYN so that it can have ASLR applied to it.
3. Modify relros.c so that it works on our static PIE executables. Inject the enable_relro() code using an infection technique that places the code into the regular text segment so that we can use an immediate call instruction as discussed in section 4.2; this will allow the standard deinitialization routines to run after our code, and after main(), in generic_start_main().

Take aways:

1. .got.plt attacks exist in statically linked executables
2. RELRO does not work with statically linked executables
3. ASLR does not work with statically linked executables
4. Some prototype solutions have been offered in this paper
5. Cleanest fix would be through gcc/ld toolchain code

Custom software presented in this paper: https://github.com/elfmaster/static_binary_mitigations

References:
[1] RELRO - https://pax.grsecurity.net/docs/vmmirror.txt

[2] ASLR - https://pax.grsecurity.net/docs/aslr.txt

[3] checksec.sh - https://github.com/slimm609/checksec.sh

[4] PAGEEXEC - https://pax.grsecurity.net/docs/pageexec.txt

[5] RANDEXEC - https://pax.grsecurity.net/docs/randexec.txt

[6] static-pie - https://gcc.gnu.org/ml/gcc/2015-06/msg00008.html

[7] static-pie patch - https://gcc.gnu.org/viewcvs/gcc?view=revision&revision=252034

[8] https://access.redhat.com/blogs/766093/posts/1975803

[9] http://phrack.org/issues/58/4.html

[10] libelfmaster - https://github.com/elfmaster/libelfmaster

[11] Relros/ASLRs - https://github.com/elfmaster/static_binary_mitigations