Position Independent Executables

Finnbarr P. Murphy

(fpm@fpmurphy.com)

Over the last few years, there has been a movement in GNU/Linux to produce more secure code and limit entire categories of malicious attacks by better use of specific hardware functionality and new compiler, linker and loader options. One of the techniques that has become commonplace is what is called Position Independent Executables (PIE).

PIE is an address space randomization technique that compiles & links executables to be position independent, i.e. machine instruction code that executes properly regardless of where in memory it actually resides. When combined with a kernel that can recognize it is loading a PIE binary, the kernel loads it into a random address instead of the traditional fixed address locations.

So how do you make a PIE binary? It is actually quite easy to do. *gcc* has included support for PIE since version 3.4. Here is the relevant except from the *gcc* man page:

```
-fpie
-fPIE
    These options are similar to -fpic and -fPIC, but generated position independent code
can be only linked into
    executables.  Usually these options are used when -pie GCC option will be used during
linking.

    -fpie and -fPIE both define the macros "__pie__" and "__PIE__".  The macros have the v
alue 1 for -fpie and 2 for
    -fPIE.
```

Other compilers have similar flags.

Consider the following trivial program (*demo.c*):

```
int local_global_var = 0x20;
int local_global_func(void) { return 0x30; }
int
main(void) {
    int x = local_global_func();
    local_global_var = 0x10;
    return 0;
}
```

which is compiled using the following to produce a PIE binary:

```
gcc -o pie -fpie demo.c
```

What are the obvious differences between a non-PIE and a PIE binary? Here *demo* is build as a regular on-PIE.

```
$ gcc -o demo demo.c
$ ldd -d -r demo
    linux-vdso.so.1 =>  (0x00007fff3bbed000)
```



```
    libc.so.6 => /lib64/libc.so.6 (0x0000003845c00000)
    /lib64/ld-linux-x86-64.so.2 (0x0000003845800000)
$ ldd -d -r demo
    linux-vdso.so.1 =>  (0x00007fffd99ff000)
    libc.so.6 => /lib64/libc.so.6 (0x0000003845c00000)
    /lib64/ld-linux-x86-64.so.2 (0x0000003845800000)
$ objump -f demo
demo:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000000000005b0
$ file demo
demo: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared
 libs), for GNU/Linux 2.6.18, not stripped
$ readelf -l demo

Elf file type is EXEC (Executable file)
Entry point 0x400390
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001c0 0x00000000000001c0  R E    8
  INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000664 0x0000000000000664  R E    200000
  LOAD           0x0000000000000668 0x0000000000600668 0x0000000000600668
                 0x00000000000001e8 0x00000000000001f8  RW     200000
  DYNAMIC        0x0000000000000690 0x0000000000600690 0x0000000000600690
                 0x0000000000000190 0x0000000000000190  RW     8
  NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x0000000000000598 0x0000000000400598 0x0000000000400598
                 0x000000000000002c 0x000000000000002c  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     8

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .
gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
$
```

Here is the same code compiled as a PIE binary:

```
$ gcc -o demo -Fpie -pie demo.c
$ ldd -d -r demo
    linux-vdso.so.1 =>  (0x00007fffe2fff000)
    libc.so.6 => /lib64/libc.so.6 (0x00007f3e73cd8000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f3e74057000)
$ ldd -d -r demo
    linux-vdso.so.1 =>  (0x00007fff053ff000)
    libc.so.6 => /lib64/libc.so.6 (0x00007f82071ac000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f820752b000)
```



```
$ obdump -f demo
demo:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400390
$ file demo
demo: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked (uses sha
red libs), for GNU/Linux 2.6.18, not stripped
$readelf -l demo

Elf file type is DYN (Shared object file)
Entry point 0x5b0
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001c0 0x00000000000001c0  R E    8
  INTERP         0x0000000000000200 0x0000000000000200 0x0000000000000200
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000894 0x0000000000000894  R E    200000
  LOAD           0x0000000000000898 0x0000000000200898 0x0000000000200898
                 0x0000000000000220 0x0000000000000230  RW     200000
  DYNAMIC        0x00000000000008c8 0x00000000002008c8 0x00000000002008c8
                 0x0000000000000190 0x0000000000000190  RW     8
  NOTE           0x000000000000021c 0x000000000000021c 0x000000000000021c
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x00000000000007cc 0x00000000000007cc 0x00000000000007cc
                 0x000000000000002c 0x000000000000002c  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     8

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .
gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .ctors .dtors .jcr .data.rel.ro .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
$
```

Note the different load addresses allocated each time *ldd* is invoked. Random load addresses help stop *return-to-libc* type attacks against functions in the main program. Note also the different results for the *file* command. An PIE executable is regarded as a dynamic shared object by the kernel and commands like *file*.

PIE is not very useful unless ASLR (Address Space Layout Randomisation) is also implemented in the kernel and the loader to randomize the location of memory allocations. Each invocation of a program that has been built with the *-pie* option will get loaded into a different memory location. ASLR has been available in the *mainline* kernel since 2.6.21. Randomizing memory allocation locations makes memory addresses harder to predict for an attacker who is attempting a memory-corruption exploit. ASLR is controlled system-wide by the value of */proc/sys/kernel/randomize_va_space*. Since the memory space layout of a process is available from */proc/*maps*, this file is made read-only except to the process itself or the owner of the process.

You can check the current ASLR setting as follows:


```
$ cat /proc/sys/kernel/randomize_va_space
```

One of the following values should be displayed:

- 0 – Disabled.
- 1 – (Conservative) Shared libraries and PIE binaries are randomized.
- 2 – (Full) Conservative settings plus randomize the start of *brk* area.

It is easy to erroneously link a position-dependent file into a PIE. This results in text relocations. When this occurs the linker will add add a *TEXTREL* flag to the PIE's dynamic section. To check for a *TEXTREL* flag in a binary:

```
$ readelf -d binary | grep TEXTREL
```

Here is a script, called *lsexec* that allow you to check, among other things, the PIE status of a single process (by name or by PID) or all the processes running on your system:

```
#!/bin/bash
# Copyright (C) 2003, 2004 Red Hat, Inc.
# Written by Ingo Molnar and Ulrich Drepper
# Updated 2008 Finnbarr P. Murphy
if [[ "$#" != "1" ]]; then
   echo "usage: lsexec [ <PID> | process name | --all ]"
   exit 1
fi
if [[ ! -f /etc/redhat-release ]]; then
   echo "ERROR: Script requires Fedora or Fedora downsteam release"
   exit 1
fi
cd /proc
printit() {
   if [[ -r $1/maps ]]; then
      printf "$(basename $(readlink $1/exe)) (PID %d) ==> " $1
      if [[ -r $1/exe ]]; then
         if readelf -h $1/exe|egrep -q 'Type:[[:space:]]*EXEC'; then
             printf "no PIE, "
         else
            if readelf -d $1/exe|egrep -q 'DEBUG[[:space:]]*$'; then
               printf "PIE, "
               if readelf -d $1/exe | fgrep -q TEXTREL; then
                  printf "TEXTREL, "
               fi
            else
               printf "DSO, "
            fi
        fi
        if readelf -l $1/exe | fgrep -q 'GNU_RELRO'; then
           if readelf -d $1/exe | fgrep -q 'BIND_NOW'; then
               if readelf -l $1/exe | fgrep -q ' .got] .data .bss'; then
                  printf "full RELRO, "
               else
                  printf "mincorrect RELRO, "
               fi
           else
               printf "partial RELRO, "
           fi
        else
           printf "no RELRO, "
        fi
      fi
```



```
      lastpg=$(sed -n '/^[[:xdigit:]]*-[[:xdigit:]]* rw..\([[:xdigit:]]*\) 00:00 0$/p' $1/
maps | tail -n 1)
      if echo "$lastpg" | egrep -v -q ' rwx. '; then
         lastpg=""
      fi
      if [[ -z "$lastpg" ]] || [[ -z "$(echo $lastpg || cut -d ' ' -f3 | tr -d 0)" ]]; the
n
          printf "execshield enabled\n"
      else
          printf "execshield disabled\n"
          for N in $(awk '{print $6}' $1/maps | egrep '\.so|bin/' | grep '^/' | sort -u)
          do
              NE=$(readelf -l $N | fgrep STACK | fgrep 'RW ')
              if [[ "$NE" = "" ]]; then
                  printf " => $N disables exec-shield!\n"
              fi
          done
      fi
   fi
}
if [[ -d $1 ]]; then
   printit $1
   exit 0
fi
if [[ "$1" = "--all" ]]; then
    for N in [1-9]*; do
      if [[ $N != $$ ]] &amp;amp;&amp;amp; readlink -q $N/exe > /dev/null; then
         printit $N
      fi
    done
    exit 0
fi
for N in $(/sbin/pidof $1); do
   if [[ -d $N ]]; then
      printit $N
   fi
done
exit 0
```

To check all running processes:

```
# lsexec –all
```

For each running process, *lsexec* will output a line of information including whether the executable is a PIE or not. Other information outputted includes whether or not Exec Shield is enabled for the process. Exec Shield is enabled if the process does not require an executable stack, disabled if it does require one or if it was build with an old version of gcc that doesn't support the executable/non-executable stack flag.

Other information outputted includes whether the process is *full RELRO*, *mincorrect RELRO*, *partial RELRO*, or *no RELRO*. RELRO is a memory area overwrite mitigation technique which moves commonly exploited structures in ELF binaries to different locations. In addition, the loader marks the relocation table as read-only for those symbols resolved at load-time. With *full RELRO*, the GOT (Global Offset Table) is made read-only after relocation by the linker.

Finally, another useful script for checking the status of things like Stack Smashing Protection (SSP), ASLR, the NX bit, RELRO, and PIE is Tobias Klein's checksec script.

