Too much PIE is bad for performance

Mathias Payermathias.payer@nebelwelt.net

Department of Computer Science, ETH Zurich

Abstract

Compiling an application as a Position Independent Executable (PIE) enables Address Space Layout Randomization to protect the application from security attacks by mapping the executable at a random memory location. Nevertheless most applications on current x86 Linux distributions are mapped to a static address for performance reasons.

This paper evaluates the overhead and side-effects of PIE using the SPEC CPU2006 benchmarks on an Intel Core i7 CPU with a recent Ubuntu distribution. Our analysis shows that the overhead for PIE on 32bit x86 is up to 26% for some benchmarks with an (arithmetic) average of 10% and a geometric mean of 9.4%. We identify the increased register pressure as the source for most of the overhead on x86.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection; D.3.4 [*Programming Languages*]: Processors — Run-time environments; D.3.4 [*Programming Languages*]: Processors — Code generation

General Terms Performance, Security, Optimization

Keywords Position Independent Executable (PIE), Linux, ASLR, Optimization, Security

1. Introduction

Software security is an important problem and current compilers and systems include different protection techniques to make exploitation harder. Protection mechanisms like Data Execution Prevention (DEP)[1] [15] that enforces either W⊕X pages, a non-executable stack, stack canaries (ProPolice [6]), and Address Space Layout Randomization (ASLR [2, 3, 8]) enable probabilistic protection from code injection and code reuse attacks.

Return-into-libc [7, 12], Return oriented programming (ROP) [10] and jump oriented programming (JOP) [4] are three modern attack techniques that no longer rely on injected executable code but reuse available application code. Both return-into-libc and ROP rely on an unchecked application stack, i.e., return addresses on the stack are not verified. Modern runtime guards (e.g., libdetox [9]) use a separate shadow stack to check return addresses and therefore prohibit return-into-libc and ROP based attacks. JOP based attacks are more complicated to run and also more complicated to protect against. A runtime system either checks the integrity of every dynamic control flow instruction [1, 5] or the compiler ensures that no open dynamic control flow instructions (e.g., `jmp *%eax`; an indirect jump through the `eax` register) are available in the compiled source. Both ROP and JOP rely on known addresses for code locations (gadgets) that are then concatenated in an actual exploit.

Address Space Layout Randomization (ASLR) [2, 3, 8] randomizes all memory regions of an application (e.g., dynamically loaded libraries, heap, and stack). A potential exploit can no longer rely on constant addresses for, e.g., library routines and gadgets. A drawback of this approach is that the address space for 32bit binaries is small and only a few bits can be randomized which opens the possibility of probabilistic attacks [13], e.g., the Linux ASLR implementation [8] only offers 16bit of entropy (according to [13]).

The Linux ASLR implementation [8] on x86 is limited if the application itself is not compiled as a Position Independent Executable (PIE). In particular non-PIE ASLR applications are mapped to the constant address `0x0804800` (this includes the `data` section, `bss` section, `code` section, `GOT` section, and `PLT` section). An application can be compiled into a PIE which can then be loaded at random addresses. Linux distributions like Ubuntu only compile a small set of binaries (27 for Ubuntu 11.10) as PIE due to a "5-10%" performance penalty according to the Ubuntu Security wiki [14]. All other programs are compiled without PIE. According to [11] the overhead for PIE in I/O based benchmarks is between 0% and 10%. Unfortunately we were unable to reproduce the results of [11] for bzip2 (0% overhead on their system) on our system.

PIE protects applications probabilistically from ROP and JOP. The current PIE implementation for GCC uses an additional register to hold the base pointer of the current module. The PIE compilation scheme increases register pressure for architectures with a small amount of registers. E.g., x86 has 8 registers whereas 6 or 7 (EAX, EBX, ECX, EDX, ESI, EDI, and depending on the compilation flags, EBP) registers can be used to hold program values. If the PIE flag is used then the number of available registers is reduced to 5 or 6 at any point in the program.

This paper evaluates the effective overhead for PIE binaries in different configurations using the SPEC CPU2006 benchmarks. The evaluation shows that the overhead is non-negligible and varies between 0.37% and 26% with a geometric mean of 9.4%.

[1] DEP uses the executable bit for pages in modern memory management units to enable non-executable data regions. DEP ensures that only code pages are executable. A stronger guarantee is $W \oplus X$ which ensures that a page is either writeable or executable but not both. Linux uses an $W \oplus X$ approach called *Exec Shield* [15].

Benchmark	-O3 [s]	-O3 -fPIE [s]	Ovhd. [%]
400.perlbench	369	463	25.47%
401.bzip2	610	713	16.89%
403.gcc	308	334	8.44%
429.mcf	254	262	3.15%
445.gobmk	479	550	14.82%
456.hmmer	554	584	5.42%
458.sjeng	533	671	25.89%
462.libquantum	560	624	11.43%
464.h264ref	760	829	9.08%
471.omnetpp	298	323	8.39%
473.astar	472	492	4.24%
483.xalancbmk	242	259	7.02%
433.milc	394	400	1.52%
444.namd	536	538	0.37%
447.dealII	426	431	1.17%
450.soplex	258	270	4.65%
453.povray	244	290	18.85%
470.lbm	327	328	0.31%
482.sphinx3	520	607	16.73%
Average	429	472	10.12%
Geo. mean	405	443	9.40%

Table 1: Performance of SPEC CPU2006 for -O3 and relative overhead for PIE.

Benchmark	-O2 [s]	-O2 -fPIE [s]	Ovhd. [%]
400.perlbench	378	462	22.22%
401.bzip2	635	768	20.94%
403.gcc	314	342	8.92%
429.mcf	264	270	2.27%
445.gobmk	478	553	15.69%
456.hmmer	556	586	5.40%
458.sjeng	561	693	23.53%
462.libquantum	570	622	9.12%
464.h264ref	782	859	9.85%
471.omnetpp	308	340	10.39%
473.astar	501	540	7.78%
483.xalancbmk	257	283	10.12%
433.milc	449	454	1.11%
444.namd	537	537	0.00%
447.dealII	481	492	2.29%
450.soplex	263	275	4.56%
453.povray	248	292	17.74%
470.lbm	329	332	0.91%
482.sphinx3	524	610	16.41%
Average	444	490	10.37%
Geo. mean	420	460	9.72%

Table 2: Performance of SPEC CPU2006 for -O2 and relative overhead for PIE.

2. Evaluation

This section shows the evaluation for the PIE feature of GCC that produces position independent executables. The PIE feature enables ASLR for binaries. The evaluation uses GCC version 4.5.2-8ubuntu on Ubuntu 11.04 with Linux kernel version 2.6.38-15-generic and glibc version 2.13. The evaluation system uses a Intel Core i7 dual core CPU clocked at 3.07 GHz with active SMP and 12GB RAM. The evaluation uses all benchmarks of the SPEC CPU2006 v1.01 benchmark suite that compile using recent GCC versions. The evaluation uses two different compilation settings. The benchmarks are compiled with either `-O3` or `-O2`. The benchmarks are executed using the `runspec` program and the configuration uses 3 runs. The CPU2006 logs are available on request.

2.1 PIE and -O3

Table 1 compares SPEC CPU2006 performance for `-O3` (the most aggressive optimization level of GCC)with and without `-fPIE`. Wee see that PIE executables are never faster than non-PIE executables and the overhead varies between 0.37% and 26% depending on the benchmark. The benchmarks can be grouped into 4 groups: negligible overhead between 0% and 2% (4 benchmarks), small overhead up to 5% (3 benchmarks), medium overhead between 5% and 10% (5 benchmarks), and high overhead with more than 10% performance penalty(7 benchmarks).

The benchmarks with high overhead either have a highly irregular workload with a large amount of indirect control flow transfers (400.perlbench and 458.sjeng) or process streams of data (401.bzip2, 453.povray, and 482.sphinx3). These workloads have a high register pressure and the reduced set of registers is the source for the high overhead.

The average overhead for PIE (when compiled with O3) is 10% and the geometric mean is 9.4%. This overall non-negligible overhead is the reason why not all applications are compiled with PIE. The Ubuntu distribution chooses performance over the increased security benefit that PIE offers.

2.2 PIE and -O2

Table 2 shows a comparison between binaries with PIE and without PIE on the `-O2` GCC optimization level. The results are comparable to `-O3` with an average overhead of 10% and a geometric mean of 9.7% for PIE.

2.3 PIE comparison

Figure 1 compares the SPEC CPU2006 results for -O2 and -O3. The benchmarks are ordered by descending overhead for -O3. The overhead for both -O2 and -O3 is comparable for all benchmarks.

2.4 PIE and x64

x64, the 64bit extension of x86 does not have the same limitations as 32bit x86. First of all, x64 doubles the number of

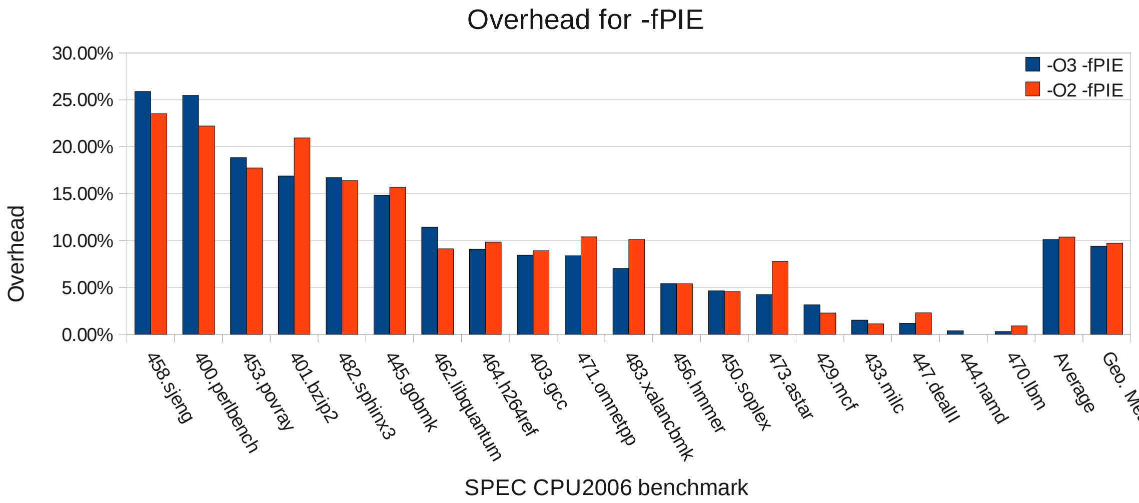

Figure 1: Comparison of the overhead for PIE between -O2 and -O3.

registers: x86 offers 16 total registers of which 15 registers can be used for computation. Secondly, x64 uses an addressing mode that is relative to the instruction pointer, thereby removing the need to use an extra register for PIE.

A quick evaluation for x64 reports an average overhead of 3.61% and a geometric mean of 2.34% for an -O3 optimization level on the same system using the "test" dataset of SPEC CPU2006.

3. Conclusion

Applications that are compiled as Position Independent Executable (PIE) increase the protection from code-reuse attacks like ROP and JOP by enabling ASLR for the application image as well (and not only for shared libraries and stack). On architectures with a small number of registers like 32bit x86 the increased security comes at a performance price of up to 26% for individual benchmarks with an average of 10% and a geometric mean of 9.4% due to increased register pressure.

Current Linux distributions do not tolerate the overhead for 'fat' PIE applications and trade (increased) security for performance. The high overhead for PIE calls for alternative security solutions that enable ROP and JOP protection at a lower cost (both for individual benchmarks as well as overall performance).

Acknowledgments

My thanks go to Per Larsen and Andrei Homescu for feedback and discussions about PIE and all the different binary formats. I would also like to thank Per Larsen for pushing me to finally publish my measurements and results as a technical report.

References

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *CCS'05: Proc. 12th Conf. Computer and Communications Security* (2005), pp. 340–353.

[2] Bhatkar, E., DuVarney, D. C., and Sekar, R. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *SSYM'03: Proc. 12th USENIX Security Symp.* (2003), pp. 105–120.

[3] Bhatkar, S., Bhatkar, E., Sekar, R., and DuVarney, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *SSYM'05: Proc. 14th USENIX Security Symp.* (2005), pp. 255–270.

[4] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS'11: Proc. 6th ACM Symp. on Information, Computer and Communications Security* (2011), pp. 30–40.

[5] Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: Software guards for system address spaces. In *OSDI'06* (2006), pp. 75–88.

[6] Hiroaki, E., and Kunikazu, Y. ProPolice: Improved stack-smashing attack detection. *IPSJ SIG Notes* (2001), 181–188.

[7] Nergal. The advanced return-into-lib(c) exploits. *Phrack 11*, 58 (Nov. 2007), `http://phrack.com/issues.html?issue=67&id=8`.

[8] PaX-Team. PaX ASLR (Address Space Layout Randomization). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[9] Payer, M., and Gross, T. R. Fine-grained user-space security through virtualization. In *VEE'11: Proc. 7th Int'l Conf. Virtual Execution Environments* (2011), pp. 157–168.

[10] Pincus, J., and Baker, B. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy 2* (2004), 20–27.

[11] Roglia, G. F., Martignoni, L., Paleari, R., and Bruschi, D. Surgically returning to randomized lib(c). In *ACSAC* (2009), IEEE Computer Society, pp. 60–69.

[12] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS'07: Proc. 14th Conf. on Computer and Communications Security* (2007), pp. 552–561.

[13] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *CCS'04: Proc. 11th Conf. Computer and Communications Security* (2004), pp. 298–307.

[14] Ubuntu. List of programs built with PIE. `https://wiki.ubuntu.com/Security/Features#pie`, May 2012.

[15] van de Ven, A., and Molnar, I. Exec shield. `https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004.