Prevention of Buffer Overflow Exploits in IA32-based Linux

Prabhaker Mateti and Sowgandh Sunil Gadi
Department of Computer Science and Engineering
Wright State University
Dayton, Ohio 45435-0001
{pmateti,sgadi}@cs.wright.edu
Speaker: Sowgandh Sunil Gadi

Abstract

We review four independent modifications, known as OWL, paged-PaX, seg-PaX patches and RSX module, made to the Linux kernel that aim to prevent buffer overflow exploits in IA32-based Linux. We show that two of these modification are ineffective, even though the ideas that they embody are workable. We bring attention to the fact that Linux on IA-32 does not use segmentation wisely. We also discuss the performance impact on the kernel.

Categories & Subject Descriptors

D.4.6 Operating Systems: Security and Protection;
K.6.5 Management of Computing and Information Systems: Security and Protection

Keywords:

Internet security, Buffer overflow, Stack Smashing, Security Exploits, Linux, IA-32, Operating Systems, Kernels, Experimentation

1 Introduction

Exploiting a buffer overflow coding error in a root-privileged program is arguably the most common attack of the decade. Run-time stack smashing attacks consist of first finding a root-privileged program that an ordinary user can invoke that is carelessly written in the following sense. The program copies user given argument strings into local variables allocated on the run-time stack. The user then cleverly constructs an argument that would overwrite the return address with that of a so-called shellcode. The shellcode results in a shell with root privileges for the attacker.

The Bazaar style open source community [9][2] of Linux implies that no one has the “authority” to insist that certain security design fixes are so important that they should be adopted even at the expense of performance. Linux kernel design is rarely explained by its authors. While the source code of Linux kernel contains numerous helpful comments (compared to those in the original Unix code), for a good understanding one turns to books written by others (e.g., [3]). Nevertheless, we expected thorough documentation by the authors of security patches in explaining their idea behind the fix, a justification that the fix does not break anything else, and the impact of the fix on performance. We feel that the attention paid to other matters of kernel design is not happening to security fixes. The peer review process is nearly absent.

We focus here on just one class of security problems, namely buffer overflow exploits, and their fixes. There are several fixes that are independent in their approaches. It is not obvious if they can all be integrated without causing inconsistencies and severe performance loss.

The claim to fame of Linux has been via the IA-32 architecture, and yet Linux kernel seriously ignores the segmentation features, and the ability to address 64 GB of physical memory. We discuss such processor architecture implications.

Section 2 summarizes the background information necessary to appreciate the rest of this paper. We then focus on four kernel modifications that aim to prevent the buffer overflow attack. We devote one section to each modification where we present our explanation of how the modification accomplishes (or not) its goals. The source code of the demonstration test programs we used is available at [7]. The source code of the demonstration test programs we used is available at [7].

2 Background

The main idea of all the patches that we consider here, except paged-PaX, is to change the address ranges of code and data segments so that they are disjoint by adjusting the GDT table and LDT tables. Corresponding changes are also made in the functions which handle memory mapping system calls mmap(), munmap(),
mremap(), mprotect(). In order to better understand these issues, we describe several background topics briefly in this section.

2.1 Buffer Overflow Exploit

In the buffer overflow attack, code is injected into stack or heap or bss address locations using essentially assignment statements or data-move operations, and control is cleverly pointed to the location of these deposited instructions so that they are executed. These are called buffer overflow exploits because there is a local variable of an array type present in the original code whose bounds are unchecked. It is also called stack smashing because the contents are deliberately, but with precision, modified. See [1] for a lucid account of this attack technique. The web site of [6] contains a version of this paper with minor errors corrected.

There are both compile-time prevention techniques (see, e.g., [5]) and execution-time prevention techniques. Using virtual memory techniques, we can make certain pages non-executable at run-time. The run time systems of modern programming languages by-and-large assume that the stored program von Neumann model of the system divides the memory space into instruction space that is executable but read-only and not modifiable, and data space that is readable, writable but not (usually) executable. However, some languages have constructs that require the generation of code at run-time. The linkage facilities are such that it is easier to place such code (so called “trampoline” code, and code for signal handling) on the run time stack make the CPU’s instruction pointer point to it.

2.2 IA-32 Segmentation

IA-32 architecture requires segmentation with optional paging [4]. The running image of a process is a collection of segments. Depending on the needs of a segment containing code, data, stack, or heap of a program, the OS is expected to assign different protection features, such as read-only, read-plus-write-but-no-execute. Even though IA-32 supports $2^{213} - 1 = 16383$ segments only six of these are accessible without reloading segment registers.

The so-called (Protected) Basic Flat Model uses four segments: one for code and another for data, in kernel and user modes, and all have the same base address of 0 and the segment size of 4 GB. Protection between operating-system and application code and data is provided by the page-level protection mechanism, not by segments. This model has become the de facto standard on all major operating systems currently running on the IA-32, including Windows NT/2000/XP, Linux, OpenBSD, and FreeBSD.

2.3 IA-32 Paging

The entries of page directories and page tables have the same structure. Each entry includes, among other things, the fields: User/Supervisor flag, and the Read/Write flag. There is no explicit flag controlling whether a page contains executable code.

Recently used page-directory and page-table entries are implicitly loaded into on-chip caches called translation lookaside buffers (TLBs) to speed up linear address translation.

2.4 General Protection Exception

The processor detects some 30 different kinds of violations by raising a General Protection Exception. Some of the violations relevant to this paper are the following. 1) Exceeding the segment limit when accessing the CS, DS, ES, FS and GS segments. 2) Exceeding the segment limit when referencing a descriptor table (except during a task switch or a stack switch). 3) Transferring execution to a segment that is not executable. 4) Writing to a code segment or a read-only data segment. 5) Reading from an execute-only code segment.

2.5 Segmentation in Linux

Linux uses the Basic Flat Model. The four segments are as shown in Table 1. All these segments are mapped to the same overlapping address space from 0x0 to 0xffffffff. The runtime stack and heap are part of the data segments.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Limit</th>
<th>mode</th>
<th>rwx</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel code</td>
<td>0x0</td>
<td>0xffffffff</td>
<td>kernel</td>
<td>r-x</td>
</tr>
<tr>
<td>kernel data</td>
<td>0x0</td>
<td>0xffffffff</td>
<td>kernel</td>
<td>rw-</td>
</tr>
<tr>
<td>user code</td>
<td>0x0</td>
<td>0xffffffff</td>
<td>user</td>
<td>r-x</td>
</tr>
<tr>
<td>user data</td>
<td>0x0</td>
<td>0xffffffff</td>
<td>user</td>
<td>rw-</td>
</tr>
</tbody>
</table>

Table 1: The GDT of Linux

2.6 Memory Maps of Processes

We are interested in examining the /proc/*/maps and /proc/*/status of processes in the standard (i.e., unpatched) kernel and the patched kernel. Below we explain the details of the bash shell as seen in a standard kernel.
In the above, perm is a set of permissions of a region \( (r = \text{read}, w = \text{write}, p = \text{shared (copy on write)}) \), and inode corresponds to a file with pathname (/bin/bash) that this region belongs to on the device given by dev (major:minor), a 0 indicates that no inode is associated with the region. A region with no inode associated is often called an anonymous or null region. The stack and bss regions are anonymous: the last line `bffa0000-c0000000 rwxp fffffff0 00:00 0` is describing the stack. A permission list of `r-xp` describes a code region, and `rw-p` describes a data region. Thus, most pathnames above have two lines. All ELF executables use the dynamic linker/loader `/lib/ld-2.2.4.so`. Most executables use the C language standard library `/lib/libc-2.2.4.so`. The exact version numbers of these two are irrelevant here. The specific values shown above for dev and inode are dependent on where the files are installed and hence are not relevant. The specific values for address space and offset depend on the versions of kernel and the process.

### 3 OWL

OWL [10] is a collection of modifications to the Linux kernel that aim to fix not only the buffer-overflow problem but others.

The limit (i.e., the size) of the user code segment is decreased to `0xbfffffffff` so that the code segment would not overlap with the first 8MB of the stack. Except for this, no other changes are made. The maps will be the same in both OWL-patched and standard kernels.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Limit</th>
<th>mode</th>
<th>rwx</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel code</td>
<td>0x0</td>
<td>0xffffffff</td>
<td>kernel</td>
<td>r-x</td>
</tr>
<tr>
<td>kernel data</td>
<td>0x0</td>
<td>0xffffffff</td>
<td>kernel</td>
<td>rw-</td>
</tr>
<tr>
<td>user code</td>
<td>0x0</td>
<td>0xbfffffffff</td>
<td>user</td>
<td>r-x</td>
</tr>
<tr>
<td>user data</td>
<td>0x0</td>
<td>0xffffffff</td>
<td>user</td>
<td>rw-</td>
</tr>
</tbody>
</table>

Table 2: The GDT of Linux, as modified by OWL

This patch prevents only stack execution. Heap and bss executions cannot be prevented by this patch. An attempt to execute an instruction located on the stack will have an address outside the code segment and throws an exception and the General Protection handler terminates the process.

However, we can easily construct programs that break the OWL patch. Even though the default stack size is 8MB, any user can increase the max stack size for his processes using the shell command `ulimit`, or invoking `setrlimit()` system call to set the system resources like stack limit, max number of files open etc.

### 4 Page Tables Based PaX

There are two patches[8] provided by the PaX group. This section discusses the patch that uses page table tricks, and leaves the GDT exactly as it would be in the standard kernel.

Control reaches the page handler in the standard Linux kernel either because a referenced page was not present or because an illegal reference was made. PaX deliberately maps the page table entries for bss, heap and stack regions of the user process with supervisor privileges, i.e., pages which ought to be accessible in the user mode are now mapped to be accessible only when the CPU is in kernel mode. So when the process, in user mode, accesses them, page faults are generated. The page fault handler provided by PaX then does extra checking. If the page fault is due to a missing page, control is transferred to the standard page handler. Otherwise, the PaX page fault handler detects if code execution is from stack, heap, or bss by examining the page fault error code, the address that caused the page fault, the current value of instruction pointer register (EIP). Exceptions which are caused by a reference to a page that legitimately belongs to the process address space, but simply denied access because of kernel privileges set by PaX, are given user privileges “temporarily”.

Upon entry to the page handler, there is a 3-bit page fault error code present on the stack top. PaX handler is interested in the page faults that occurred when in user mode because of privilege violation; the error code for such faults is 101 or 111. The handling of all other faults is left to the original page fault handler.

Of the page faults with error code 101, page faults that occurred when processor fetches data because of PaX setting supervisor privileges for bss, heap and stack addresses are artificially caused by PaX. So the Supervisor/User flag of the page table entry of the address of this page fault is set to user privileges. Then processor resumes execution, no page faults occurs with the address this time and so the page table entry of the address is loaded into translation look-aside buffers. Once this address gets loaded into the translation buffer, the page table entry of the address in the page tables is restored to PaX-normal state – i.e., the Supervisor/User flag is set to supervisor privileges. But the page table entry of the address in translation buffer will have the flag set to user privileges. So the data in that address location is accessible without any page fault as long as the page table entry of the address remains in DTLB.

Of the page faults with error code 111, the page faults that occurred when processor attempts to write are caused because PaX deliberately set supervisor privileges on stack, heap, bss. The treatment is the same as above with error code 101.
argv[1] user sys pfpatched pfstd pfpax
1 0.00 0.00 686 354 332
2 0.00 0.00 942 354 588
3 0.01 0.01 1200 354 846
257 0.02 0.05 66478 354 66124
100000 5.71 17.86 25600786 351 25600435

Table 3: PaX Kernel 2.4.18

PaX generates page faults for every access to a unique address in stack, heap, bss accessed by a process. So the performance suffers. A simple test program paxtest.c executed on Pentium 3 processor demonstrates the effect of PaX on performance. The following are obtained from the output of time command: user = user time in seconds, sys = system time in seconds. The pfpatched column shows the total number of page faults in the PaX-patched kernel, and pfstd shows the total number of faults in the standard kernel, pfpax is the number of extra faults occurred because of PaX. These page fault numbers were obtained by our own patches.

5 Segmentation Based PaX

The seg-PaX patch that uses segmentation and non-obvious memory mapping of various ELF regions. The GDT of a kernel patched with seg-PaX is as follows. Note that the user data segment and user code segment are disjoint. Any attempts made by attacker to fetch code from stack or heap or bss are detected and prevented in the pagefault handler. No extra page faults are created because of this patch.

<table>
<thead>
<tr>
<th>segment</th>
<th>base</th>
<th>limit</th>
<th>mode</th>
<th>rwx</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel code</td>
<td>0x0</td>
<td>0xffffffff</td>
<td>kernel</td>
<td>r-x</td>
</tr>
<tr>
<td>kernel data</td>
<td>0x0</td>
<td>0xffffffff</td>
<td>kernel</td>
<td>r-w</td>
</tr>
<tr>
<td>user code</td>
<td>0x6000000</td>
<td>0x5fffffff</td>
<td>user</td>
<td>r-x</td>
</tr>
<tr>
<td>user data</td>
<td>0x0</td>
<td>0x5fffffff</td>
<td>user</td>
<td>r-w</td>
</tr>
</tbody>
</table>

Table 4: The GDT of Linux, as modified by seg-PaX

Seg-PaX maps all regions (text, data, bss and stack) in data segment just it is done in a standard kernel. It also maps corresponding anonymous region, in code segments at the same offset as that of text region in data segment. Anonymous regions in code segment and text regions in data segment are backed by the same physical memory frames. Text, data and bss regions of bash are mapped from 0x08048000. Dynamic linker and libraries are mapped from 0x20000000. Stack is mapped from 0x60000000 towards lower addresses. In the user code segment, for every text region in data segment there is a corresponding anonymous region.

When the process begins, instructions are fetched from code segments and since they are all anonymous regions pagefaults occur. Let us assume that addr contains the address of an instruction in the code segment which caused the page fault. addr_m contains the address of the page in data segment corresponding to the addr in code segment.

First the page table entries for pages at addr and addr_m are prepared. The required pages of the file are loaded into the physical memory. The page table entries of the page at addr_m are filled with physical addresses at which file is loaded. The page table entries of page at addr are also filled with that in the entries of the page at addr_m. As a result, the page table entries of page at addresses addr and addr_m both point to the same physical address. The pagefault handler returns, and processor resumes execution until it reaches the next page of code segment where the pagefault re-occurs and the above is repeated.

6 RSX

Unlike the previous patches which make changes to several lines in the source code of the kernel requiring recompilation, RSX [11] is built as a loadable kernel module that can be dynamically loaded into a standard kernel.

While compiling RSX module we have an option of selecting either GDT or LDT. If GDT is selected, then segment selector for CS register is 0x3b and if LDT is selected then its is 0xf. RSX fills the seventh entry of the GDT table, which is unused in the standard kernel, with a new descriptor for user code segment with base at 0x50000000, and the limit 0x6fffffff. The range of data segment is unchanged. RSX allocates new LDT for the process and fills the first entry of the LDT table, with the seventh GDT descriptor.

RSX maps all regions (text, data, bss and stack) in data segment just as in a standard kernel. A text region is mapped to both data segment and code segment at the same offset. However, the mapping does not permit a process to modify its text, and processes which require
modifiable text regions will crash in a kernel loaded with RSX module. Unlike seg-PaX, RSX makes modifications only to the handler of `mmap()`. No modifications are made in other memory map functions.

Any attempt made by an attacker to fetch code from stack is detected and prevented in the general protection error handler. Suppose the attacker has overflowed a buffer at `0xbeffffff` and injected shell code. Then the return address is pointed to `0xbeffffff`. When the function returns, the next instruction the process has to execute is at an offset of `0xbeffffff` in the code segment whose base address is `0x50000000`. But the limit of code segment is `0x6fffffff` which is less than `0x0xbeffffff` raising a general protection error.

Heap and bss execution are prevented in the page fault handler. The address of the shell code becomes the offset in the code segment whose base address is `0x50000000`. When the function returns, the address of the next instruction is `0x50000000 + address of shell code`. Nothing is mapped at that address and so it is regarded as bad address and page fault handler deals with it.

In an RSX-loaded kernel, the total size of virtual memory of a process is limited to `0xc0000000-0x50000000`. The total size of virtual memory of `bash` process under RSX-loaded kernel is `4584KB` but under standard kernel it is `2636KB` because each text region is mapped at different addresses, once in data segment from `0x0` and again from `0x50000000`. Total physical memory allocated for `bash` process in RSX-loaded kernel is `1628KB` where as in standard kernel it is `1540KB` for the same reason.

7 Conclusion

We have not been able to fully trace the reasons for why Linux designers chose to use the Basic Flat Model of segmentation, but the following are possible reasons. Loading segment registers requires several memory cycles. System calls implemented via the `INT` instructions, applicable only when using Basic Flat Model, are faster.

Proper use of IA-32 segments would prevent a large class of buffer overflow exploits. But Linux makes minimal use of segments.

Suppose we were able to detect an attempt to buffer overflow. What should be the action that kernel takes? Kill the offending process at the risk of causing a denial of service attack? We wish to add this as an exception that the programmer would handle.

When we began the study of these patches, we expected it to require deep concentration, but found that the code is poorly documented. It is not sufficient that the source code listings of programs are open to public view. It is necessary that proper documentation and design rationale are also made available. Often the author of a program is not excited about these. But others should be encouraged and rewarded for filling in this gap.

References


