Address Space Layout Randomization Lab

Copyright © 2014 Christopher Marron, University of Maryland Baltimore County.
Adapted from *Address Space Layout Randomization Lab*, available at
`http://www.cis.syr.edu/~wedu/seed/all_labs.html`.
Copyright © 2006 Wenliang Du, Syracuse University.
The development of this document is funded by the National Science Foundation's Course, Curriculum, and Laboratory Improvement (CCLI) program under Award No. 0618680 and 0231122. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation. A copy of the license can be found at http://www.gnu.org/licenses/fdl.html.

1 Overview

Address space layout randomization (ASLR) is a computer security technique which involves randomly arranging the positions of key data areas in a process' address space. These key data areas usually includes the base of the executable and position of libraries, heap, and stack. Although ASLR does not eliminate vulnerabilities, it can make the exploitation of some vulnerabilities much harder. For instance, a common buffer-overflow attack involves loading the shellcode on the stack and overwriting the return address with the starting address of the shellcode. In most cases, attackers have no control over the starting address of the shellcode: they have to guess the address. The probability of a successful guess can be significantly reduced if the memory is randomized.

2 Memory Layout in Minix 3

Important Note: After Minix 3.1.0, significant changes were made to the process management service. This lab *must* be completed using Minix 3.1.0.

The Process Manager (PM)'s process table is called `mproc` and its definition is given in

`/usr/src/servers/pm/mproc.h`.

The process structure defined in `mproc.h` contains an array `mp_seg` which has entries for the text, data and stack segments respectively. Each entry consists of three variables storing the virtual address, physical address and length of the segment. Minix 3 programs can be compiled to use either the **combined I and D space** (Instruction and Data), where the system views the data segment and the text segment as one BIG segment or **separate I and D space**. Combined I and D spaces are necessary for certain tasks like bootstrapping or for cases in which a program needs to modify its own code. By default all the programs are compiled to have Separate I and D spaces. Figure 1 shows a process in memory (OS independent).

When a program is compiled to have a common I and D space, the text segment is always empty and the data segment contains both the text and the data. This is a security vulnerability. The system no longer differentiates between the two segments so an attacker may be able to load malicious machine code in the data segment and make the system execute it. This is particularly bad if the program runs with elevated privileges (e.g. `setuid root`). The memory layout for a combined I and D space is shown in Figure 2.

A program, when compiled to have separate I and D space, will have non-zero text and data segments. This is done not as a security measure but for efficiency reasons: having separate I and D segments allows many instances of the same program to share the same text segement. The memory layout for separate I and D spaces is represented Figure 3.

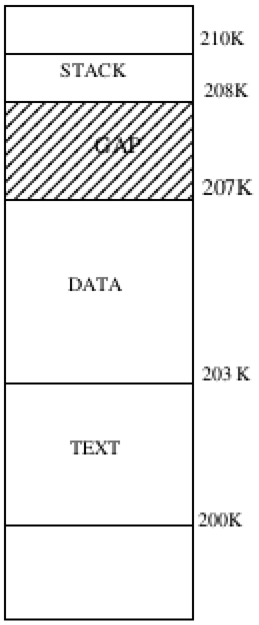

Figure 1: A process in memory

	Virtual	Physical	Length
Stack	0x8	0xd0	0x2
Data	0	0xc8	0x7
Text	0	0xc8	0

Figure 2: Memory map (`mem_map`) for combined I and D

Given a virtual address and a space to which it belongs, it is a simple matter to see whether the virtual address is legal or not, and if legal, what the corresponding physical address is.

2.1 `exec()` system call

Once a program is compiled, it must be loaded into memory to execute. The `exec()` system call handles this process. The `exec()` call performs several steps:

1. Check Permissions – is the program file executable?
2. Get the segment and the total sizes.
3. Fetch the arguments and the environment from the caller.
4. Allocate new memory and release un-needed old memory.
5. Copy the stack to the new memory image.
6. Copy the data (and maybe text) segment to the new memory image.
7. Handle setuid/setgid bits.

	Virtual	Physical	Length
Stack	0x5	0xd0	0x2
Data	0	0xcb	0x4
Text	0	0xc8	0x3

Figure 3: Memory map (`mem_map`) for separate I and D

8. Fix-up process table entry.
9. Tell the kernel that the process is now runnable.

In order to randomize the starting address of a variable on the stack, we need to introduce some level of randomness in step 4 or 5. Randomizing the gap space (figure 1) in a way that does not effect the execution of a process is a possible approach.

2.2 `malloc()` library call

`malloc()` is used to allocate memory from the heap. It causes the data segment to expand into the lower memory region of the gap area (while the stack eats away the top portion). `malloc()` invokes the `_brk()` call, which in turn calls `do_brk()`, causing the data segment to grow. `do_brk()` also checks if the data segment is colliding with the stack segment. If all the conditions are satisfied, the data segment increases by the amount of memory requested, with adjustments made to ensure that the segment ends on a word boundary. The address of a heap area requested by `malloc()` can easily be randomized by `malloc`-ing a small random sized fragment after `exec`-ing the process or before `malloc`-ing for the first time.

3 Lab Task

Your task is to modify Minix 3 to randomize the locations of variables on the stack and the heap. We have mentioned two areas of the Minix 3 source code where you should consider making modifications:

1. In `/usr/src/servers/pm`, the file `exec.c` defines the TEXT, DATA, and STACK segments and allocates memory. The file `mproc.h` defines the `mproc` structure that stores the locations of segments for each process.
2. In `/usr/src/lib/ansi`, the file `malloc.c` defines the `malloc()` function for allocating additional heap space.

You will need to generate random numbers, and you may use either of two random number generators available in Minix 3: `rand()` or `random()`.

3.1 Testing your modifications

The following program creates a variable on the stack and another on the heap and prints their addresses:

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include< alloca.h>
```

```
int main(int argc, char *argv[])
{
 int onStack;
 int *onHeap = (int *) malloc(sizeof(int));
 printf("Starting Stack at %x \nStarting Heap at %x\n", &onStack, onHeap);
 free(onHeap);
 return 0;
}
```

If you have randomized the location of variables on the stack and heap correctly, the program should print different memory addresses on each execution.

4 Submission

Your final submission will consist of a written report and soft copy of modified source files. The written report must include a general description of your approach, descriptions of the specific modifications made, and evidence that the modifications work correctly (this could be a screen capture of several executions of the test program given above). The report must be well-written, using correct spelling and grammar. The source files provided must be sufficient for me to build your modified version of Minix 3 and test it myself, if necessary.

In addition to the final submission, you must submit two milestone assignments. Milestones will not be graded for writing quality but must, of course, be understandable:

Milestone 1 Written report describing your proposed implementation of ASLR for the stack. The report must include a draft version of your stack ASLR code (include code excerpts within the report).

Milestone 2 Written report describing your proposed implementation of ASLR for the heap. The report must include a draft version of your code heap ASLR code (include code excerpts within the report).