Address Space Layout Randomization Lab

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's address space. These key data areas usually includes the base of the executable and position of libraries, heap, and stack, etc. Although ASLR does not eliminate vulnerabilities, it can make the exploit 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 success guess can be significantly reduced if the memory is randomized. Students need to implement ASLR for Minix 3.

2 Memory Layout in Minix3

The 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 three entries for text, data and stack segment respectively. Each entry in turn has another three entries storing the virtual address, the physical address and the length of the segment. Minix3 programs can be compiled to use either the **combined I and D space** (Instruction and Data) space, 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 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 the an attacker can load a corrupt assembly on the data segment and make the system execute it(which thinks that its a text code). The memory layout for a combined I and D space takes the following form:

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

The program, when compiled to have separate I and D space, have non-zero text and data segments. This was however not done for security reasons but for efficiency. Minix3 does not support paging(or virtual memory) and is targeting to be an embedded OS. Having separate I and D has an added advantage of efficiency. Many instances of the same program can share the same text segement. The memory layout for separate I and spaces is represented as follows:

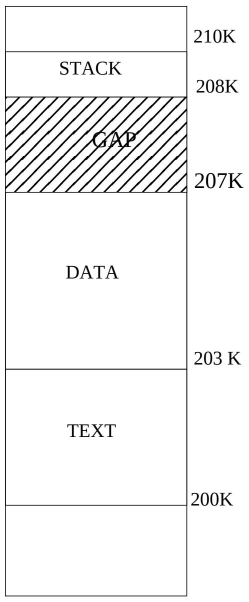

Figure 1: A process in memory

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

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.

The program once compiled needs to be loaded into the memory. The `EXEC` system call takes care of that.

2.1 EXEC system call

The exec() call does its job in the following steps:

1. Check Permissions- Is the file executable?
2. Get the segment and the total sizes.
3. Fetch the argument and the environment from the caller.
4. Allocate new memory and release unneeded old memory.
5. Copy stack to new memory image.
6. Copy data(and maybe text) segment to new memory image.
7. Handle setuid/setgid bits.
8. Fix up process table entry.

9. Tell the kernel that the process is now runnable.

If we need to randomize the starting address of a variable on stack, then we need to introduce some level of randomness in step 4 or 5. Randomizing the gap space (figure 1) in a way that it does not effect the execution of a process might be one way to do so.

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()) which causes 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 (adjustments are made so that it lies on a word boundary). The address of a heap area requested by malloc() can easily be randomized by mallocing a small random sized fragment after execing the process or before mallocing for the first time.

3 Lab Task

This lab expects the students to randomize the stack and the heap. You may use the existent rand() or the random() functions provided by the C library. Consider the following program:

```
#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 \n Starting Heap at %x\n",&onStack,onHeap);
 free(onHeap);
 return 0;
}
```

3.1 Stack Randomization

The above program should print a different value of the stack on each execution. `/usr/src/servers/pm` would be good place to insert your code in.

3.2 Heap Randomization

Heap randomization assures that the starting address of the heap is different for each execution. You would need to modify the malloc() library call defined in `/usr/src/lib/ansi` to achieve this.

4 Submission and Demonstration

You should submit a detailed lab report to describe your design and implementation. You should also describe how you test the functionalities and security of your system. You also need to demonstrate your system to us. Please sign up a demonstration time slot with the TA. Please take the following into consideration when you prepare for demonstraiton:

- The total time of the demo will be 15 minutes, no more additional time would be given. So prepare your demonstration so you can cover the important features.
- You are entirely responsible for showing the demo. We will NOT even touch the keyboard during the demonstration; so you should not depend on us to test your system. If you fail to demo some important features of your system, we will assume that your system does not have those features.
- You need to practice before you come to the demonstration. If the system crashes or anything goes wrong, it is your own fault. We will not debug your problems, nor give you extra time for it.
- During the demo, you should consider yourself as salesmen, and you want to sell your system to us. You are given 15 minutes to show us how good your system is. So think about your sales strategies. If you have implemented a great system, but fail to show us how good it is, you are not likely to get a good grade.
- Do turn off the messages your system prints out for debugging purposes. Those messages should not appear in a demonstration.