Sources of Non-Determinism in Program Execution

Syed Raza

April 7, 2011

1 Introduction

Take a single program, and execute it several times. Does the program always execute the same sequence of instructions? If not, what can cause multiple executions of the same program to diverge in behavior?

One can immediately find obvious reasons for multiple executions of the same program to be different: instructions in multi-threaded applications can be interleaved in several ways; inputs or data can differ; random number generators are frequently used, and so on. Surprisingly, however, even a simple "Hello, world!" application in C does not always produce the same execution trace.

This paper describes sources of non-determinism – obvious and obscure – for certain classes of programs, along with strategies to make program execution deterministic. Such a study of application behavior is novel and interesting in its own right; it reveals important interactions between applications, underlying operating systems and intermediate software layers, and provides insight into common application behavior.

2 Terminology

In this paper, an application's *execution trace* refers to instructions executed, in user mode, by the process for the application, along with the side-effects of those instructions (e.g. values read/written from registers or memory).

Two execution traces can thus be different if they contain different instructions (e.g. control flow is not identical), or if the same instructions have different side effects (e.g. application memory is not identical).

We use dynamic binary instrumentation tools like DynamoRio and Pin to collect execution traces from applications, and to modify application behavior to overcome any sources of non-determinsm.

3 Before entering `main()`

This section explains why execution traces can diverge even before programs have started executing. All these factors combine to explain why even a simple "Hello, World!" program in C can generate many different traces.

3.1 Address Space Layout Randomization (ASLR)

Implemented by several mainstream operating systems, Address Space Layout Randomization (ASLR) is a security feature which involves random arrangement of the positions of key data areas for an executing program. For instance, the base of the executable, the position of libaries, heap and the stack in a process's address space can be different every time the program is run.

ASLR hinders several kinds of security attacks because it makes it difficult for attackers to predict addresses for redirecting execution. For instance, *return-to-libc* attacks usually start with a buffer overflow attack which replaces the return address on the stack with the address of another existing instruction and overwrites an additional portion of the stack to provide arguments to this function. Because the common C runtime on UNIX systems ("libc") is always linked with programs, and contains useful calls (e.g. `system()`) to run arbirtrary programs, most attacks simply provide one argument on the stack and execute a libc function like `system()`. If libc functions are always placed at the same addresses, attackers can avoid injecting malicious code, and just redirect execution to them.

Non executable stacks cannot prevent return-to-libc attacks. While stack-smashing protection can help prevent such attacks, ASLR is the most effective in making return-to-libc attacks very difficult on 64-bit machines. On 32-bit machines, new attacks reduce the effectiveness of ASLR to searching from a 16-bit space, which can be done in a matter of minutes.

Because of ASLR, two execution traces of a simple "Hello, World!" C program are entirely different (i.e. they have no overlap). Almost every instruction has a different address, and stack addresses are clearly different too.

Because ASLR is not effective in preventing return-to-libc attacks on 32-bit machines, ASLR can be disabled to make program execution more deterministic. Most distributions of Linux allow ASLR to be turned off via the following command:

```
sudo kernel.randomize_va_space=0
```

Disabling ASLR greatly increases the overlap between different traces of the "Hello,World!" program, but the traces are not identical yet.

3.2 Stack Protection: `libc` Canary

3.3 Pointer Encryption

3.4 Environment Variables

3.5 Elf Auxiliary Vectors

3.6 Command Line Arguments

3.7 Thread Local Storage

4 Inside `main()`

4.1 `rdtsc` instruction

4.2 `cpuid` instruction

4.3 `getpid()` system call

4.4 reads from `/dev/urandom`

4.5 reading the current time