Cloaking Order in Chaos

Invisibly subverting the Linux random number generator via the hypervisor

Checkpoint

Jeremy Erickson (jericks) *Andrew Quinn (arquinn)* *Timothy Trippel (trippel)*
EECS 588, Winter 2016
University of Michigan, Ann Arbor

1 Project Overview

Our project is predicated on the assumption that a Nation-State Actor (NSA) may decide to focus on subverting the Random Number Generator (RNG) of virtual machines through control of the hypervisor.

2 Progress

We have conceptualized several approaches by which the NSA could approach this task. The first is to simply insert breakpoints in the kernel at the memory locations where random numbers are returned to the user, and return pseudo-random NSA-controlled random numbers in their place. The second approach is to introspect on any memory read from the entropy pool, and overwrite the entropy pool with a pseudo-random NSA-controlled value immediately before the read, thus making the resulting random bytes predictable to the NSA. The third approach is to control the sources of entropy to the entropy pool itself. There may be other approaches to this goal that we have not considered, such as making significant modifications to the running kernel in memory to deterministically generate pseudo-random numbers without the need for breakpoints, but we believe these other approaches to be unlikely.

Of these three approaches, we chose to implement the first, as it is the most straightforward and gives the attacker the most control over the output of the random number generator.

2.1 Attack Prototype

Our attack prototype subverts the Linux random number generator of a virtual machine running on top of the Xen hypervisor. Our approach to do so has been based on the first approach above, described by Alt et al. [2], and is detailed in Figure 1. To do so, we have utilized LibVMI [1], a virtual machine introspection tool, to implement a software hook (or software breakpoint event) on any accesses to `/dev/random` and `/dev/urandom`. This software hook replaces all bytes requested from either `/dev/random` or `/dev/urandom` interfaces with attacker determined bytes, by overwriting the user space buffer passed to the *extract_entropy_user* function. Because the interfaces to `/dev/random` and `/dev/urandom` both invoke a call to the *extract_entropy_user* function, a single software hook successfully subverts both interfaces, shown in Figure 1 as "LibVMI Hook Location 1" and "LibVMI Hook Location 2". Upon reverse engineering how the Linux entropy pool management system works, we discovered that it is possible to intercept calls to the input entropy pool as well. This may be useful for a slightly modified attack where the attacker controls the inputs to the blocking and nonblocking entropy pools themselves rather than the out of the entire random number generator interfaces. This is denoted "LibVMI Hook Location 3" in Figure 1.

2.2 Detection

Our approach creates hypervisor breakpoints on specific offsets within the *extract_entropy_user* function in the Linux kernel's RNG. This approach works as long as the offsets that the attack is using correspond to the particular version of the Linux kernel that the victim is using. However, making simple changes to the Linux kernel's `random.c` file will change these offsets and cause any existing attack to fail. We consider it unlikely that the NSA's attack would be sophisticated enough to dynamically profile the kernel's RNG and detect the changed offsets with a high enough confidence factor to proceed with the attack automatically.

The attacker has two options: First, the attacker could decide to take a hash of each kernel before she tries to attack the kernel. If the hash differs from a known version the attacker can simply choose not to continue with the

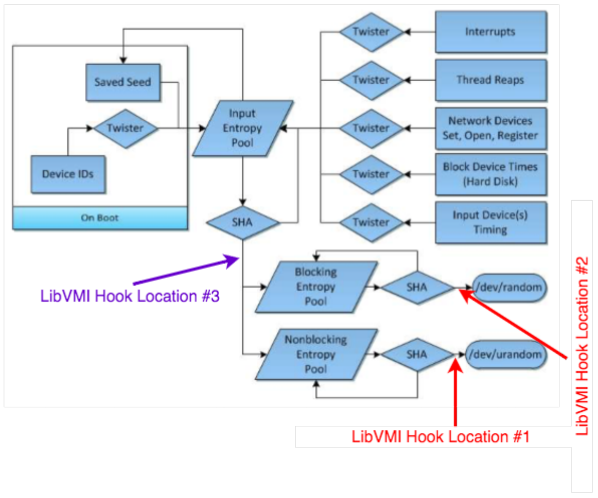

Figure 1: Diagram of the Linux entropy pool management architecture (diagram courtesy of Alt et al. [2])

attack. In this case, a very simple detection scheme leverages the difference in timing between a compromised hypervisor and a non-compromised hypervisor. A victim simply starts multiple machines, a few with a stock version of the Linux kernel and a few with small changes to `random.c` (i.e. a couple of monitoring calls to `printk`). The victim then determines the amount of time it takes to read a large number of bytes from `/dev/urandom` in each of the kernels. By repeating this process the victim will be able to tell if there is a statistical difference between the amount of time it takes to read random bytes in the stock kernel compared to the amount of time it takes to read in the modified kernel. This detection scheme is not unique to this particular attack; it is likely that we could detect any of the attack vectors we have presented using this scheme.

An especially sophisticated NSA may choose to dynamically determine which offsets to hook when presented with a new kernel. The NSA could do so by disassembling the kernel binary and analyzing the object files. As long as the custom kernel has the same basic layout in the `random.c` file, it is likely that the process could even be automated with some confidence factor. To detect the attack in this case, a victim simply has to find a way to leak the values that come from the entropy pool before the attacker can change them. A simple scheme leaks these bytes using `printk` and periodically compares them against the bytes returned from the RNG. By unpredictably changing the operation of the Linux RNG, any previously-deployed attack will be forced to either operate on an unknown kernel, thus likely crashing it, or refuse to operate, thus allowing a timing attack to reveal its existence.

3 Schedule

We are currently on schedule. We have completed our initial attack prototype, and can overwrite the return value from the Linux RNG with arbitrary bytes. Between now and April 21, we plan to:

1. Update our attack prototype to return pseudo-random bytes that are predictable to the attacker but otherwise appear completely random.

2. Implement a modified Linux kernel that can detect our prototype attack by comparing the entropy pool state against the output random bytes.

4 Obstacles and Workarounds

Our first attempt focused on the KVM hypervisor. Unfortunately, LibVMI does not support *events* (e.g. catching interrupts or memory accesses) when run on the KVM hypervisor and we need *event* support to efficiently manage breakpoints in the VM. Therefore we changed course to attack the Xen hypervisor. Due to a hardware issue, we wasted several weeks at the beginning of the semester trying to get the Xen hypervisor to detect Dom0's hard drive.

We are not currently blocked by any other obstacles that we are aware of.

5 Preliminary Results

We have done a brief initial investigation into the feasibility of detecting a timing difference between benign and malicious hypervisors. So far, the results look very promising. Over 10 iterations of copying a single 512-byte block from `/dev/urandom`, a benign hypervisor allows the operation to complete in a mean of 3.6 milliseconds with a standard deviation of 0.49 milliseconds. Our prototype attack incurs a comparatively huge performance penalty, with a mean of 19.1 milliseconds and a standard deviation of 1.29 milliseconds. This performance impact initially appears to be linear with the size of the random bytes generated.

References

[1] *LibVMI: Virtual Machine Introspection*. Apr. 1, 2016. URL: `http://libvmi.com/`.

[2] Matthew Alt et al. *Entropy Poisoning from the Hypervisor*. Unpublished class project. 2015. URL: `https://courses.csail.mit.edu/6.857/2016/files/alt-barto-fasano-king.pdf`.