Entropy Poisoning from the Hypervisor

6.857 Final Project

Matthew Alt
William Barto
Andrew Fasano
Andre King

May 13, 2015

Abstract

The increasing prevalence of virtualization increases the relevance of security for guest operating systems. An important facet of system security is cryptography, which relies on secure random number generation. Three approaches to compromising a guest operating system's random number generation from a hypervisor are presented. The first approach, directly modifying generated random numbers in memory, has been successfully demonstrated with a proof-of-concept. A backdoored architectural random number generator as it applies to recent version of the Linux kernel is presented, with an additional proof-of-concept. Finally the possibility of manipulating hardware timings and inputs to poison the guest operating systems random number generation is analyzed.

1 Introduction

Modern cryptography relies on secure random numbers; however, computers are deterministic and, therefore, predictable. Random numbers are used to generate cryptographic material to prevent potential attackers from predicting private cryptographic keys. Insufficient randomness has been shown to be a weak point for networked computers in the past [14, 6].

The use of cloud services and other virtualization technology is on the rise. Gartner estimates that "at least 70% of i386 server workloads are virtualized [15]." The virtualizer controls the hardware for all virtualized guests. A malicious virtualizer, or hypervisor, could potentially control the random numbers generated in a virtualized operating system. By doing so, malicious hypervisors could compromise the cryptographic security of their guest operating systems.

This research demonstrates proof-of-concept attacks on a guest operating system's random number generation from within a hypervisor. While this should serve as a reminder of the importance and implications of trusting one's hardware (and cloud services), this research could also be used to improve cyber security defenses. By implementing the techniques described here within a malware sandbox, such as Fireeye's[5], security researchers could reduce the cryptographic complexity of malware samples, aiding their malware analysis.

2 Background

2.1 Hypervisors

Hypervisors are software that support emulation of systems, allowing a host system to run a virtual machine. These virtual machines can range from small embedded systems to full operating systems[7]. Hypervisors serve many purposes including running multiple operating systems on a single machine, isolating processes for security reasons, and cloud computing[7]. Xen[3], VMware[17], and QEMU[13] are all example hypervisors in use today.

QEMU was chosen for this research effort, because it is open source and the PANDA source code provides some insight into how QEMU handles non-determinism[4]. QEMU supports cross platform emulation, so multiple platforms can be emulated on a single host. QEMU handles cross platform support by converting all emulated machine instructions to an intermediate language called Tiny Code Generator (TCG). TCG can then be converted to the machine language used by the host system. Sections of code that do not result in control flow changes are generated into translation blocks and converted as a set. QEMU can be run with KVM[9] or Xen acceleration, where QEMU does not perform the translations and simply allows the other virtualizer to handle the instruction more quickly. Without hardware acceleration QEMU is emulating the guest's hardware in software that executes on the host system. Acceleration is not used for this project.

2.2 Randomness

Random numbers can be generated in many ways, from physical methods, such as coin flipping and die rolling, to electronic methods, such as camera noise[18] and quantum noise[8]. Due to the deterministic nature of software, when truly random numbers are needed, the randomness is derived from interactions with hardware. Different Operating Systems (OS) handle random number generation differently and some processors now contain hardware random number generators[11].

The Linux kernel's random number generation is centered around a variable called the entropy pool. The specifics of kernel's random number generation vary from kernel to kernel, but generally a seed from the entropy pool is saved to disk when the system is shutdown and read from disk to seed the entropy pool when the system is started. Various device identifiers, such as MAC address, are also mixed into the entropy pool on boot. The process of "mixing" bytes into the pool is described in Section 2.2.2. As the system is running other sources of presumably unpredicatable data are mixed into the entropy pool. For the kernel inspected, the utilized unpredictable data sources are: interrupt timings, thread reaps, network device timings, block device timings, and input device timings. Figure 1 shows the general process by which the entropy pool is updated. When random bytes are requested from the kernel, the entropy pool is hashed using SHA and the results of the hash are returned. The one-wayness of the hashing function prevents an attacker from determining the state of the entropy pool by requesting random numbers. The output of this hash is also mixed back into the entropy pool. This prevents an attacker who has discovered the contents of the entropy pool from computing previous states of the entropy due to the one-wayness of the hashing function.

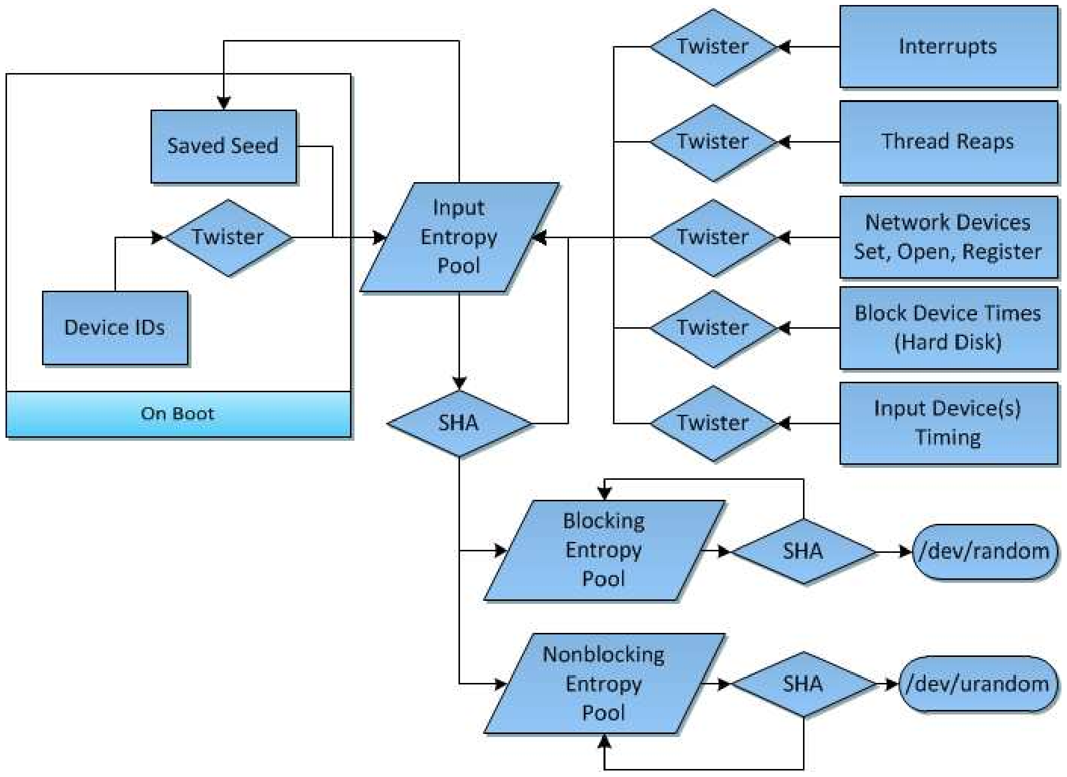


Figure 1: Linux Kernel Entropy Pool Management

2.2.1 Entropy Pools

The Linux Kernel uses three different entropy pools to manage random data. When an event occurs that can be used to increase the entropy of the system, information about this event is mixed into the input entropy pool. The other two entropy pools are known as the blocking and nonblocking pools. The nonblocking pool will continue to provide output when the estimated entropy is insufficient for cryptographic purposes. The blocking pool will wait to provide output until sufficient entropy is available. These pools typically contain little entropy until an application requests random data from one of them. When this occurs, the pool requests data of the desired length from the input entropy pool. This data is extracted from the input entropy pool, hashed using SHA and then mixed into the relevant pool using the mixing function described below. The output of SHA is also mixed back into the input pool [10].

2.2.2 Mixing

When certain events occur, the kernel adds information about this event into the input entropy pool using a mixing function. The mixing function uses the state of the entropy pool and b bytes of new data to update b bytes in the entropy pool.

One of the main events used as a source of entropy are system interrupts. When interrupts are fired, the kernel uses the system time plus data from the interrupt itself to mix the entropy pool. When this happens, a structure containing a long and two unsigned integers is populated with information about the interrupt: *jiffies*, *cycles*, and *num*. Where *num* is a number specific to the interrupt. For example, when an interrupt is triggered by a keyboard event, this value is populated with the key code exclusive-or'ed with a constant value representing the keyboard model and manufacturer.

The bytes contained in this structure are mixed into the input entropy pool one at a time. The mixing function, shown in Figure 2, uses the values in the entropy pool (shown at the top of the figure), the input byte, and a twist table populated with constant values to generate a new value for the entropy pool. The entropy pool structure maintains an index pointing to the location that should next be updated.

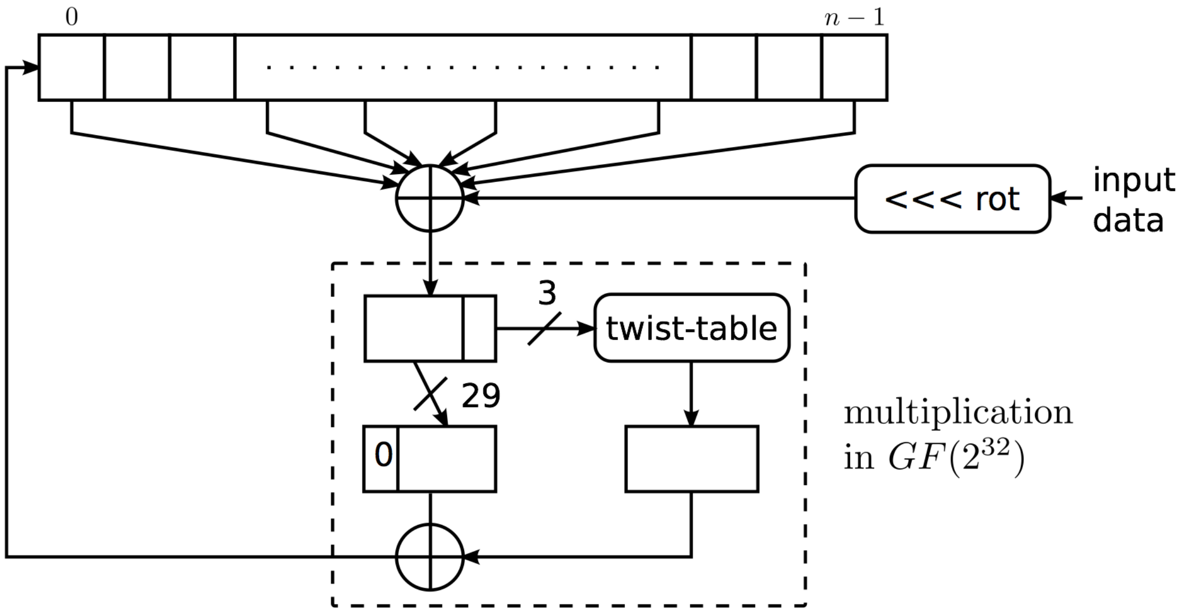


Figure 2: The mixing function in the Linux kernel. From [10]

3 Approaches

For this project three potential approaches are evaluated for manipulating the entropy pool of a guest operating system from the hypervior. The first method, which is referred to as "Intercept on Access," involves setting breakpoints in the guest operating system using the modified hypervisor. These breakpoints are set to trigger on functions that are critical to the random number generator in the Linux kernel—such as *get_random_bytes* and *random_read* in drivers/char/random.c. With this approach, the hypervisor breaks on functions of interest within the guest and modifies the return values to acheive the desired effect within the guest.

The second approach involves emulating a backdoored architectural random number generator (ARNG), in the case of the i386 architecture this would control the output from the assembly instruction *rdrand*. Instead of providing a legitimate ARNG, the hypervisor can provide a malicious random number generator by modifying the code which handles instruction parsing and TCG generation. With this malicious ARNG, the hypervisor would have complete control over the output of the *rdrand* instruction on i386, allowing corruption of the guest random number generation.

The third approach involves directly modifying the emulated hardware interrupts used as input to the guest entropy pool. If the hypervisor is sufficiently intelligent, it could produce hardware timings that compromise the guest entropy pool.

3.1 Intercept on Access

The first approach implemented involves the use of breakpoints for inspecting and modifying various functions within the Linux kernel. Breakpoints on i386 are implemented by the instruction *int 3*. When this instruction is executed on a Linux system it sends a "SIGTRAP" signal to the current process. In order to trigger this instruction, a debugger will inject an *int 3* instruction. This will cause the kernel to send a "SIGTRAP" signal to the process that is being debugged. Once the signal has been sent, the debugger receives the signal that the child process was stopped. From this point the user can use the debugger to inspect the process being debugged. For initial testing the GNU Debugger (GDB) was used to debug the target operating system running in QEMU and to set breakpoints at various points of interest. QEMU supports using GDB to debug the guest operating system by passing the flag "-S". In addition to providing basic debugging primitives, GDB has the ability to import debugging symbols. During initial testing the target Linux kernel was built with debug symbols providing more debugging capability.

During the initial testing GDB was used to modify the values returned by the functions *read_random* and *extract_buf* in the kernel. This granted the ability to directly manipulate the values that were output by the random number generator located at */dev/random*. GDB was required to make this approach work. QEMU was modified to remove the GDB dependency, because the goal was to modify the guest operating system using only the hypervisor and no additional programs.

When breakpoints are reached in the modified hypervisor, a callback function is executed. This allows the hypervisor to examine it's current state (current program counter, register values, etc.) and determine what modifications need to be made. This callback function is used to modify the return values for the functions that were originally inspected with GDB.

3.2 Backdoored Architectural Random Number Generator

The affects of backdooring the Architectural Random Number Generator (ARNG) differ depending on the version of the guest operating system. Three versions of the Linux kernel are considered here. The three snippets of code referenced in this section are included in the Appendix[16], note the function *arch_get_random_long* places the output of the ARNG into a buffer, *v*.

In the first case, Kernel Version 3.12—and any prior kernel version that supported an ARNG—, the output of the *extract_buf* function is directly controlled by the output of that ARNG. As the code shows, the entropy pool is hashed, then the output of the hash is directly exclusive-or'ed with the ARNG output. The result of that exclusive-or is then returned. If the

ARNG is backdoored, it could be modified to return the contents of the entropy pool exclusive-or'ed with the desired output of *extract_buf*. This would result in the backdoored ARNG having direct control on the output of *extract_buf*, and by the transitive property, */dev/random*.

In the second case, Kernel Versions 3.13 and 3.14, the entropy pool is exclusive-or'ed with the ARNG before it is output. This does not give an attacker arbitrary control on the output of *extract_buf*, because SHA is one-way. However, it does allow an attacker to produce predictable values, since SHA is deterministic and an attacker knows the input to the hash function.

In the third case, the current Kernel Version, the ARNG output is used as the initialization vector for the hash function. The pool is then hashed and the output of that hash is used as the output. In this case a backdoored ARNG only allows an attacker to control a subset of the input to the hash function. This means an attacker does not have direct control over the input or output of the hash function. The approach described in Section 3.1 could be used to directly control the input or output of the hash function, but if that approach is utilized controlling the output of *extract_buf* directly is simpler.

3.3 Control Entropy Sources

For the purposes of analysis, the *add_timer_randomness* function is used as it encapsulates the issues that occur across most entropy sources. Also, if the guest is a server, *add_timer_randomness* is the primary source of entropy. This function effectively has three input values: jiffies, cycles, and interrupt numbers. Jiffies are a Linux kernel internal timekeeping value, they effectively track the number of CPU timer interrupts that have occurred. Cycles is simply a measure of the number of clock ticks that have occurred, in the case of the i386 kernel, this is read using the *rdtsc* instruction. The final input value is an encoding of the interrupt that generated the event.

There are many possible techniques to control the sources of these inputs. As the number of cycles is measured using the *rdtsc* instruction, this could be controlled by editing a hypervisor's handler for *rdtsc* such that it returns augmented values while executing *add_timer_entropy*. The observability of this modification depends largely on how intelligent the modified *rdtsc* handler is—if it never returns the correct value, unrelated programs are likely to break. Controlling the other two input sources proves more challenging. Control over jiffies requires augmenting when CPU timer interrupts are delivered, which has far-reaching effects in the kernel's timing and scheduling. Possibly even more difficult, controlling the interrupt numbers (without repeating the "Interrupt on Access" solution) requires editing the interrupts delivered by the hypervisor's virtual CPU which, while possible, completely changes the semantics of the guest operation. Worse, CPU timer interrupts are one type of interrupt, and as previously discussed, editing the timing of these particular interrupts has far-reaching effects.

Supposing direct control over these sources was obtained, the values are run through a twisting function before being put into the entropy pool. Because of this, a method for inverting the *mix_pool_bytes* function was also created, which provides target values for the three inputs discussed earlier to produce a desired entropy pool. In cases where complete control over the inputs is obtained, this function could potentially be used to leverage complete control over the entropy pool.

4 Methodology

4.1 Testing Environment

For initial testing, a minimal Linux kernel with a small number of userspace applications was desired. In order to accomodate this, a custom Linux image was built then stripped of unnecessary userspace applications and utilities. Initial testing was primarily based on kernel level components, reducing build time. In order to test and boot a Linux kernel in QEMU, the following components must be present:

A kernel compiled for the target architecture.

A filesystem that can be mounted by the kernel on boot.

The filesystem that is mounted by the kernel must provide the tools necessary to probe and test the various kernel components. In order to provide necessary utilities to the guest operating system the BusyBox[2] toolkit was used. BusyBox is a set of tools that provides lightweight replacements for standard UNIX utilities such as cat, dd, and hexdump[2]. In addition to BusyBox, the ability to run standard C programs was also desired for testing purposes. This capability was added by modifying the BusyBox build process to include standard GCC (GNU Compiler Collection) tools when building the BusyBox environment. This allowed for one to cross compile programs on the host operating system and run them on the guest OS within QEMU. The resulting filesystem could also be mounted in the host OS to allow for copying files to and from the guest OS while the guest was offline, i.e. not running in QEMU.

Initially each of these components was built seperately and then combined in order to test within QEMU. After the Buildroot toolchain was discovered, it was used for all further testing. Buildroot is a set of tools that can be used to build entire Linux images for target systems [1]. Buildroot will compile and build all necessary tools for the guest operating system as well as the kernel of the guest operating system itself and outputs both a kernel and filesystem image. The resulting filesystem and image can be run in QEMU.

After using the Buildroot toolchain to build the image and filesystem, QEMU is invoked with the following command:

```bash
#!/bin/bash
qemu-system-i386 -kernel buildroot/output/images/bzImage \
-append "root=/dev/sda" \
-hda buildroot/output/images/rootfs.ext2 \
-monitor stdio \
-s \
-S
# The arguments can be broken down as follows:
# -kernel: Points to target kernel
# -append: Tells QEMU which initialization arguments to provide
#  the kernel, this tells it to append a root device
# -hda: Add a drive to the hypervisor, which will be mounted by
#  the kernel upon boot
# -monitor: Provides access to the QEMU monitor, aiding introspection
# -s: Open a GDB server on localhost:1234
# -S: Pause the guest on startup
```

4.2 Intercept on Access

For the “Intercept on Access” approach, QEMU was modified to add first-class support of breakpoints without the use of GDB. This added functionality was then used to place breakpoints in the entry and exit points of the *extract_buf* function found in *drivers/char/random.c* in the Linux kernel source code. When this function is executed, the callback checks to see if the value returned should be poisoned. At present, in the proof-of-concept, the value is modified if a custom “poison” command was previously run to provide a new set of bytes to return. However, this check could easily be made more elaborate to check things such as which processes are currently running, which process made the request, or any other set of selectors. To perform the entropy poisoning inside the breakpoint callbacks, QEMU modifies the memory of the guest to overwrite the buffer that contains the guest-computed random bytes with values selected by the hypervisor. Again, in the proof-of-concept, the bytes will be whatever argument was passed to the “poison” command, but could easily be selected by a more complex method left to the reader’s imagination.

This method of poisoning guest entropy has some great advantages and some unfortunate artifacts. The random bytes are modified by directly editing the guest’s memory, so the hypervisor has arbitrary targeted control over the guest’s kernel random number generation. This is more useful than just recording the random numbers generated if, say, there is some complicated multi-stage analysis framework which can be greatly optimized if the guest generates a particular random number. In addition, if some analysis is being performed on a program with behavior dependant on sampling the entropy pool, forcing these samples to desired values can make the analysis much easier (i.e. if encryption key is some function of random bytes and host-keyed data the random bytes may be poisoned to induce a fixed encryption key). Using this technique, poisoning entropy only for a specific user or specific applications is also trivial – whereas other, more passive, techniques modify the entropy pool for the entire system making such targeting impossible.

4.3 Backdoored Architectural Random Number Generator

The proof-of-concept for the backdoored ARNG was implemented by adding support for the associated i386 assembly instruction, *rdrand*, to QEMU. Previously, QEMU returned CPU status bits indicating an ARNG was not available and would fail to execute the command with an invalid instruction exception if an attempt was made to execute the opcode. The 30th bit of the register *ECX* returned by *CPUID* is used to indicate the presence of *rdrand* in the i386 Instruction Set Architecture (ISA). By modifying QEMU to set this bit in *ECX* when *CPUID* is called, the guest OS will believe that an ARNG is present and *rdrand* is an available instruction. Further modifying the TCG generator in QEMU to parse the *rdrand* opcode and arguments, set the specified return register to an arbitrary value, and set the appropriate status flags, allows the hypervisor to arbitrarily control the output of the ARNG presented to the guest. Intel’s supplied ARNG library, librand[11], was used for verifying compliance of the emulated instruction with the i386 standard. Modifications to Intel’s ARNG are tested using using an Arch Linux live CD booted in QEMU. Arch was chosen instead of the minimal Linux kernel used elsewhere, because Arch supports *rdrand* and quick installation of the build tools necessary to compile and run Intel’s librand library. The QEMU modifications described in this section will only work for the i386 ISA, but similar modifications should be possible for other ISAs with ARNGs.

4.4 Control Entropy Sources

A method to compute the required inputs to *mix_pool_bytes* to produce desired output bytes into the kernel’s input entropy pool was developed and is included in the Appendix. As the inputs to the *mix_pool_bytes* function are controlled by hardware, it was originally hypothesized that an attacker would be able to arbitrarily set these inputs and directly control the state of the entropy pool. The values mixed into the entropy pool are the current jiffies, the current number of cpu cycles, and an integer whose value encodes the type of interrupt which occurred.

Controlling the number of CPU cycles reported by QEMU has been implemented and tested—requiring modifications to *include/qemu/timer.h*. It is possible to exert some control over the *jiffies* value by augmenting the manner in which CPU timer interrupts are delivered.

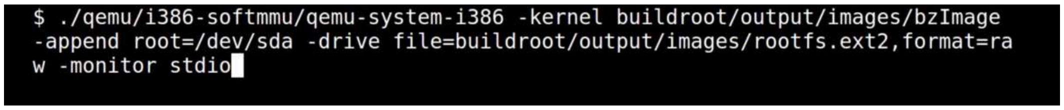
This approach has not yet been tested as it introduces even larger artifacts for slightly increased control surface – since the CPU interrupt timers must still fire for the kernel to work there will always be less-controlled interrupts tainting the poisoned pool.

The mixing function used to add unpredictable data into the input entropy pool is an invertible function. A proof-of-concept was developed that will invert this function assuming arbitrary control of the inputs. In practice, only the number of CPU cycles can reasonably be controlled. Controlling this provides partial control over the kernel's input entropy pool. However, the output of this function is later passed through a one-way function based on SHA before it is used to populate the blocking or nonblocking pools [10]. Since finding targeted collisions with SHA is computationally infeasible, the attack described above appears to be infeasible [12].

5 Results

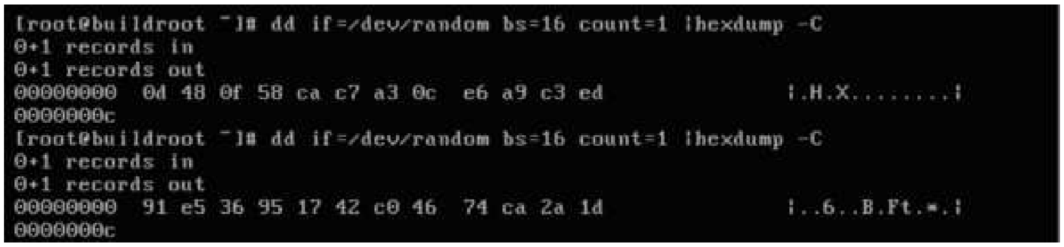
The "Intercept on Access" approach was successfully implemented as described in Section 4.2, and tested with the simplified kernel derscribed in Section 4.1. The QEMU command used for testing the proof-of-concept is shown in Figure 3. Figure 4 shows the output of */dev/random* before the poisoning is initiated. The modified QEMU hypervisor can be set to intercept the output of */dev/random* with arbitrary input. Figure 5 shows the command being used with two different values. Figure 6 shows the output of reading from */dev/random* before and after the "poison" command is invoked.

In practice, the poisoning could be done to return data that seems random to the guest. One such function could return the output of a pseudo random number generator with a known seed every time */dev/random* is accessed. This would easily allow an attacker to efficiently compute all outcomes of */dev/random* in sequence, while the guest's random number generator would appear random.



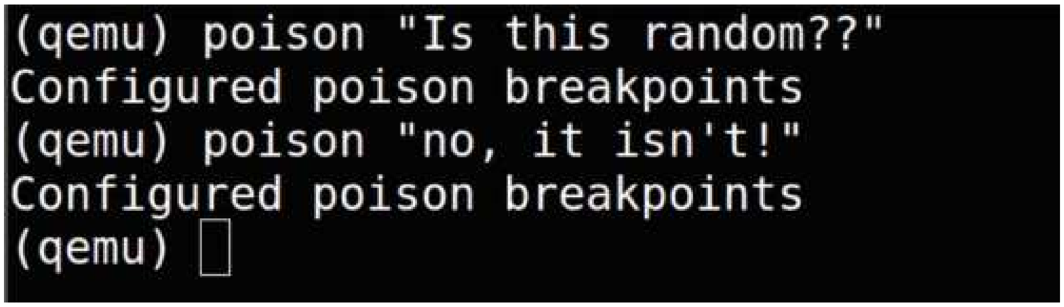
```
$ ./qemu/i386-softmmu/qemu-system-i386 -kernel buildroot/output/images/bzImage
-append root=/dev/sda -drive file=buildroot/output/images/rootfs.ext2,format=ra
w -monitor stdio
```

Figure 3: QEMU Command



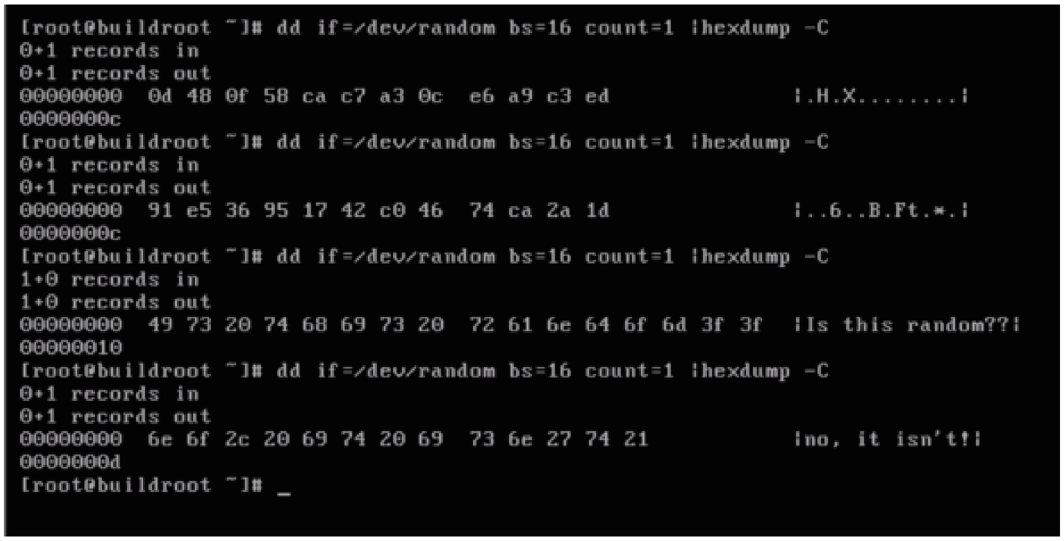
```
[root@buildroot ~]# dd if=/dev/random bs=16 count=1 |hexdump -C
0+1 records in
0+1 records out
00000000  0d 48 0f 58 ca c7 a3 0c  e6 a9 c3 ed              |.H.X........|
0000000c
[root@buildroot ~]# dd if=/dev/random bs=16 count=1 |hexdump -C
0+1 records in
0+1 records out
00000000  91 e5 36 95 17 42 c0 46  74 ca 2a 1d              |..6..B.Ft.*.|
0000000c
```

Figure 4: Output of /dev/random Before Poisoning



```
(qemu) poison "Is this random??"
Configured poison breakpoints
(qemu) poison "no, it isn't!"
Configured poison breakpoints
(qemu) 
```

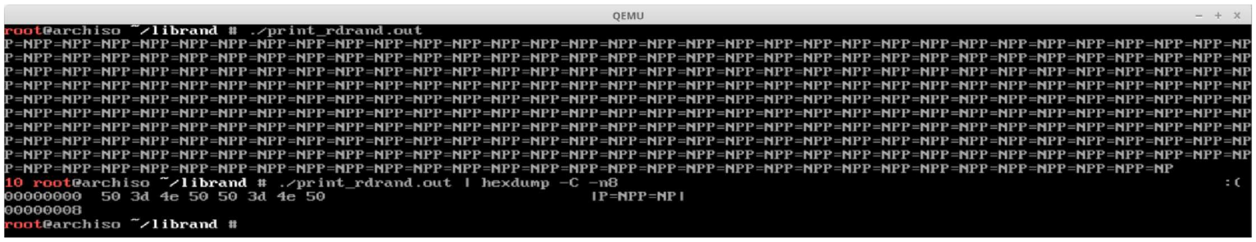
Figure 5: Issuing the Poison Command in QEMU



```
[root@buildroot ~]# dd if=/dev/random bs=16 count=1 |hexdump -C
0+1 records in
0+1 records out
00000000  0d 48 0f 58 ca c7 a3 0c  e6 a9 c3 ed              |.H.X........|
0000000c
[root@buildroot ~]# dd if=/dev/random bs=16 count=1 |hexdump -C
0+1 records in
0+1 records out
00000000  91 e5 36 95 17 42 c0 46  74 ca 2a 1d              |..6..B.Ft.*.|
0000000c
[root@buildroot ~]# dd if=/dev/random bs=16 count=1 |hexdump -C
1+0 records in
1+0 records out
00000000  49 73 20 74 68 69 73 20  72 61 6e 64 6f 6d 3f 3f  |Is this random??|
00000010
[root@buildroot ~]# dd if=/dev/random bs=16 count=1 |hexdump -C
0+1 records in
0+1 records out
00000000  6e 6f 2c 20 69 74 20 69  73 6e 27 74 21           |no, it isn't!|
0000000d
[root@buildroot ~]# _
```

Figure 6: Output of /dev/random After Poisoning

The modifications described in Section 4.3 to backdoor QEMU's i386 ARNG have been successfully implemented. Figure 7 shows the output of calling a modified version of Intel's ARNG test, included in their librand library, to print the output of the *rdrand* instruction. As the figure shows, the ARNG consistently returns $0x503D4E50$, which is the ASCII representation of "P=NP". The changes to QEMU support returning any arbitrary value from the *rdrand* instruction. In practice, an attacker could use the current state of the entropy pool to compute the return values of *rdrand* in such a way that */dev/random* outputs any desired value. The required additional step of finding and reading the state of the entropy pool from the hypervisor was not tested with the ARNG but was demonstrated in the previous proof-of-concept.



```
root@archiso ~/librand # ./print_rdrand.out
P=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NP
P=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NP
P=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NP
P=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NP
P=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NP
P=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NP
P=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NP
P=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NP
P=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NP
P=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NPP=NP
10 root@archiso ~/librand # ./print_rdrand.out | hexdump -C -n8                :(
00000000  50 3d 4e 50 50 3d 4e 50                           |P=NPP=NP|
00000008
root@archiso ~/librand #
```

Figure 7: Screenshot Displaying Backdoored ARNG

While an inverter to the mixing twister used for the input pool in the Linux kernel was successfully implemented, an end-to-end proof-of-concept was not successfully implemented for the approach described in Section 4.4. Each time the mixing function was called three values were mixed into the entropy pool. One of the three values was backdoored, *cycles*, but no practical method was found for the other two values. Additionally, no inverter was found to undo the operators used to move data from the input entropy pool to either of the output pools. Since the "Intercept on Access" approach was implemented, it could be used to control the inputs for the other two values, but inverting the one-way hash function is still not feasible. This is impractical, especially since the "Intercept on Access" approach can be used by itself to arbitrarily control the output of the guest random number generator.

6 Conclusions

When targeting Linux kernel versions 3.12 or before, using a backdoored ARNG will quickly and easily provide complete control over random numbers generated by the kernel. When targeting Linux Kernels 3.13 or 3.14 using a backdoored ARNG will allow an attacker to control the input to a one-way hash that outputs the random numbers generated by the kernel. If the Linux kernel is a more recent version, the "intercept on access" technique will be best approach to arbitrarily control the output of the kernel's random number generation. While initial research was unable to demonstrate a technique for using the entropy sources in hardware to take complete control of the kernel's entropy pool, two thirds of the entropy pool was compromised, reducing the state space in the entropy pool from 256^{128} bits to 256^{44}. Future work may reveal additional techniques to further reduce the state space of this pool.

7 Future Work

Poisoning guest entropy sources from the hypervisor has thus far proven largely unfruitful in the Linux kernel, but it may be the case that the issues encountered in recent versions of the Linux kernel do not show up in other operating systems or previous versions of the Linux kernel. Investigating other operating systems to identify entropy gathering techniques that can be reasonably controlled by hypervisor modifications without pervasive effects on the guest is a potential avenue for future research. As taining guest entropy by augmenting the sources seems infeasible without considerable artifacts, future work would likely focus mostly on breakpoint-based solutions. With breakpoint-based solutions considering all sources of entropy, even those not generally available to virtualized servers, becomes much easier as no particular set of inputs should be any more difficult to interpose on.

Adding a framework for generating seemingly random output with minimal state storage is also a goal of future work, this would allow an external analysis framework to perfectly duplicate the guest's entropy pool (or compute future values) with minimal state storage. Using a psuedo-random function and recording a per-guest seed is a candidate solution − ideally a psuedo-random function which can generate portions of it's output out-of-sequence.

Making the system resillient to guest operating system updates and supporting more guest operating systems is another area of future work. At present, the breakpoint system depends on debug symbols generated as a side-effect of the Linux kernel's build process to obtain the locations of symbols to place breakpoints, meaning a kernel patch or other source of symbol relocation would break the system. For most targets this is an acceptable dependence as debug symbols for popular Linux distributions are readily available and similarly kernel debugging symbols for other operating systems, such as Microsoft Windows, are also readilly available. Investigating other techniques to locate the code of interest could add patch resilliance to the system. In addition to adding resillience into the system, teaching it to recognize when the guest is unknown would prevent the system for injecting stray breakpoints into unknown guests which would be a highly observable artifact in a cloud hosting system.

8 Summary

Three methods for controlling the entropy of a guest system from a malicious hypervisor were investigated. The first method, "Intercept on Access" uses a customized version of QEMU to set breakpoints and modify the system memory whenever the *read_random* kernel function is called. This method gives the hypervisor complete control of the kernel's random number generation, but could be detected by the guest. A second method investigated focused on using the hypervisor to emulate an architectural random number generator and control the values it presents to the guest operating system. Until recent patches to the Linux kernel, this would have allowed for arbitrary control over the output from the kernel's random number generator. A final method of directly controlling entropy sources was investigated as well. This method was hypothesized to be feasible since the Linux kernel adds new bytes to its entropy pool using a function that can be inverted. A function was created to invert the kernel's mixing function, *mix_pool_bytes*, but initial research suggests that using modified hardware alone to directly control the input to this mixing function may be infeasible.

References

[1] Buildroot. Documentation `http://buildroot.uclibc.org/docs.html`. [Online; accessed 2-April-2015].

[2] BusyBox. About busybox `http://www.busybox.net/about.html`. [Online; accessed 2-April-2015].

[3] Chisnall, D. *The definitive guide to the xen hypervisor.* Pearson Education, 2008.

[4] Dolan-Gavitt, B., Leek, T., Hodosh, J., and Lee, W. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 839–850.

[5] FireEye. Debunking the myth of sanbox security `https://www2.fireeye.com/debunking-the-myth.html`. [Online; accessed 19-March-2015].

[6] Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A. Mining your ps and qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium* (2012), pp. 205–220.

[7] Hennessy, J. L., and Patterson, D. A. *Computer architecture: a quantitative approach.* Elsevier, 2012.

[8] Jennewein, T., Achleitner, U., Weihs, G., Weinfurter, H., and Zeilinger, A. A fast and compact quantum random number generator. *Review of Scientific Instruments 71*, 4 (2000), 1675–1680.

[9] KVM. Kernel based virtual machine `http://www.linux-kvm.org/page/Main_Page`. [Online; accessed 6-May-2015].

[10] Lacharme, Patrick, e. a. The linux pseudorandom number generator revisited. In *International Association for Cryptologic Research* (2012).

[11] Mechalas, J. Intel digital random number generator (drng) software implementation guide `https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide`. Online; accessed 3-May-2015.

[12] NIST. Secure hash standard (shs). FIPS PUB 180 - 4, 2012. [Online; accessed 21-May-2015].

[13] QEMU. Qemu open source processor emulator `http://wiki.qemu.org/Main_Page`. [Online; accessed 19-March-2015].

[14] Ristenpart, T., and Yilek, S. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS* (2010).

[15] Thomas Bittman, Mark Margevicius, P. D. Magic quadrant for x86 server virtualization infrastructure `https://www.gartner.com/doc/2788024?srcId=1-2819006590`. Online; accessed 3-May-2015.

[16] Torvalds, L. linux/drivers/char/random.c `https://github.com/torvalds/linux/blob/4f671fe2f9523a1ea206f63fe60a7c7b3a56d5c7/drivers/char/random.c`. [Online; accessed 6-May-2015].

[17] VMware. Virtualization `http://www.vmware.com/virtualization/`. [Online; accessed 6-MAy-2015].

[18] Zhang, X., Qi, L., Tang, Z., and Zhang, Y. Portable true random number generator for personal encryption application based on smartphone camera. *Electronics Letters 50*, 24 (2014), 1841–1843.

Appendix A Linux Kernel Source: extract_buf [16]

A.1 Kernel Version 3.12

```
sha_init(hash.w);
spin_lock_irqsave(&r->lock, flags);
for (i = 0; i < r->poolinfo->poolwords; i += 16)
    sha_transform(hash.w, (__u8 *)(r->pool + i), workspace);

/*
 * We mix the hash back into the pool to prevent backtracking
 * attacks (where the attacker knows the state of the pool
 * plus the current outputs, and attempts to find previous
 * ouputs), unless the hash function can be inverted. By
 * mixing at least a SHA1 worth of hash data back, we make
 * brute-forcing the feedback as hard as brute-forcing the
 * hash.
 */
__mix_pool_bytes(r, hash.w, sizeof(hash.w), extract);
spin_unlock_irqrestore(&r->lock, flags);

/*
 * To avoid duplicates, we atomically extract a portion of the
 * pool while mixing, and hash one final time.
 */
sha_transform(hash.w, extract, workspace);
memset(extract, 0, sizeof(extract));
memset(workspace, 0, sizeof(workspace));

/*
 * In case the hash function has some recognizable output
 * pattern, we fold it in half. Thus, we always feed back
 * twice as much data as we output.
 */
hash.w[0] ^= hash.w[3];
hash.w[1] ^= hash.w[4];
hash.w[2] ^= rol32(hash.w[2], 16);

/*
 * If we have a architectural hardware random number
 * generator, mix that in, too.
 */
for (i = 0; i < LONGS(EXTRACT_SIZE); i++) {
    unsigned long v;
    if (!arch_get_random_long(&v))
        break;
    hash.l[i] ^= v;
}
memcpy(out, &hash, EXTRACT_SIZE);
```

A.2 Kernel Versions 3.13 and 3.14

```
sha_init(hash.w);
spin_lock_irqsave(&r->lock, flags);
for (i = 0; i < r->poolinfo->poolwords; i += 16)
    sha_transform(hash.w, (__u8 *)(r->pool + i), workspace);

/*
 * If we have a architectural hardware random number
 * generator, mix that in, too.
 */
for (i = 0; i < LONGS(20); i++) {
    unsigned long v;
    if (!arch_get_random_long(&v))
        break;
    hash.l[i] ^= v;
}

/*
 * We mix the hash back into the pool to prevent backtracking
 * attacks (where the attacker knows the state of the pool
 * plus the current outputs, and attempts to find previous
 * ouputs), unless the hash function can be inverted. By
 * mixing at least a SHA1 worth of hash data back, we make
 * brute-forcing the feedback as hard as brute-forcing the
 * hash.
 */
__mix_pool_bytes(r, hash.w, sizeof(hash.w), extract);
spin_unlock_irqrestore(&r->lock, flags);

/*
 * To avoid duplicates, we atomically extract a portion of the
 * pool while mixing, and hash one final time.
 */
sha_transform(hash.w, extract, workspace);
memset(extract, 0, sizeof(extract));
memset(workspace, 0, sizeof(workspace));

/*
 * In case the hash function has some recognizable output
 * pattern, we fold it in half. Thus, we always feed back
 * twice as much data as we output.
 */
hash.w[0] ^= hash.w[3];
hash.w[1] ^= hash.w[4];
hash.w[2] ^= rol32(hash.w[2], 16);

memcpy(out, &hash, EXTRACT_SIZE);
```

A.3 Kernel Version 3.15

```
sha_init(hash.w);
for (i = 0; i < LONGS(20); i++) {
    unsigned long v;
    if (!arch_get_random_long(&v))
        break;
    hash.l[i] = v;
}

/* Generate a hash across the pool, 16 words (512 bits) at a time */
spin_lock_irqsave(&r->lock, flags);
for (i = 0; i < r->poolinfo->poolwords; i += 16)
    sha_transform(hash.w, (__u8 *)(r->pool + i), workspace);

/*
 * We mix the hash back into the pool to prevent backtracking
 * attacks (where the attacker knows the state of the pool
 * plus the current outputs, and attempts to find previous
 * ouputs), unless the hash function can be inverted. By
 * mixing at least a SHA1 worth of hash data back, we make
 * brute-forcing the feedback as hard as brute-forcing the
 * hash.
 */
__mix_pool_bytes(r, hash.w, sizeof(hash.w));
spin_unlock_irqrestore(&r->lock, flags);

memzero_explicit(workspace, sizeof(workspace));

/*
 * In case the hash function has some recognizable output
 * pattern, we fold it in half. Thus, we always feed back
 * twice as much data as we output.
 */
hash.w[0] ^= hash.w[3];
hash.w[1] ^= hash.w[4];
hash.w[2] ^= rol32(hash.w[2], 16);

memcpy(out, &hash, EXTRACT_SIZE);
```

Appendix B Inverse Mix

```
uint32 un_twist(uint32 desired_out) {
  unsigned short twist_index = 0; // Could use any index 0-8
  uint32 twist = twist_table[twist_index];
  uint32 undo_twist = ( (desired_out ^ twist) << 3 ) + twist_index;
  return undo_twist;
}

int compute_byte(uint32 *pool_cpy, unsigned long taps[5], int wordmask,
        uint32 desired_out) {
  uint32 i, undo_xors, this_xor_index, this_xor,val;
  undo_xors = un_twist(desired_out);

  for( i=0; i < 5; i++ ) {
        this_xor_index = (next_index + taps[i]) & wordmask;
        undo_xors = undo_xors ^ pool_cpy[this_xor_index];
  }
  uint32 nxt = pool_cpy[next_index];
  uint32 undo_rotate = (undo_xors ^ nxt) >> current_input_rotate;

  if (next_index==0) current_input_rotate += 7;
  current_input_rotate = (current_input_rotate+7) & 31;

  pool_cpy[next_index] = desired_out;
  next_index = (next_index -1) & wordmask;
  return undo_rotate;
}

int untwist_wrapper(struct entropy_store *r, uint32 *desired_vals) {
  uint32 i, input_vals[sizeof(desired_vals)];
  unsigned long taps[5];
  taps[0] = r->poolinfo->tap1;
  taps[1] = r->poolinfo->tap2;
  taps[2] = r->poolinfo->tap3;
  taps[3] = r->poolinfo->tap4;
  taps[4] = r->poolinfo->tap5;

  uint32 *pool_cpy = malloc(sizeof(uint32) * r->poolinfo->poolbytes);
  int wordmask = r->poolinfo->poolwords - 1;
  next_index = r->add_ptr;
  current_input_rotate = r->input_rotate;

  for (i = 0; i < sizeof(desired_vals); i++ ) {
        input_vals[i] = compute_byte(pool_cpy, taps, wordmask,
                desired_vals[i]);
  }
}
```