Cloaking Order in Chaos

Subverting the random number generator via the hypervisor

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

Abstract

In the modern domain of computing, Nation State Adversaries (NSAs), are often characterized as having the most resources, researchers, and legal authority. We investigate a hypothetical attack vector an NSA could use to gain widespread access to modern communications and web services. We present the scenario in which an NSA has obtained superuser privileges on a fraction of the host machines of a cloud service provider. Specifically, we investigate how an NSA can subvert a virtual machine's Random Number Generator (RNG) to produce deterministic outputs via the hypervisor.

In this paper we describe our attack prototype against the Linux 4.4.6 kernel. Our attack subverts reads from `/dev/random` and `/dev/urandom` and allows an attacker to produce a deterministic byte stream. We extend this attack to work against user space RNGs, specifically the OpenSSL RNG which is used by modern web servers such as Apache2 and NGINX. Finally, we describe a detection scheme for the Linux RNG and discuss how we can extend the scheme to work against this class of attacks.

1 Introduction

In the computer and network security domain, the Nation State Actor (NSA), is typically characterized as the strongest adversary imaginable. With nearly an unlimited amount of computing resources, funding, intelligent cryptographers and engineers, and the legal authority to get access to almost any information they need, this adversary has been able to use its strategic advantages to create sophisticated attacks which have subverted several modern day computing and communication systems [3, 5, 6].

In this project we investigated how an NSA could leverage its resources and abilities to gain widespread access to modern communications and web services while avoiding public criticism through secrecy and stealth. Specifically, we focused on a hypothetical scenario in which an NSA has, through exploitation, coercion, or another method, acquired superuser privileges on some fraction of the host machines of a cloud service provider (i.e. has access to the hypervisor), such as those operated by Amazon (AWS [12]), Microsoft (Azure [16]), and Google (Compute Engine [13]).

Despite its power, we make one critical assumption about an NSA that we believe is reasonable given the existing political and legal climate:

> An NSA must perform offensive actions in a manner of utmost stealth. Detection of offensive actions by any non-NSA personnel will lead to a full investigation and removal of superuser privileges on the cloud provider, as well as damage to public opinion.

Under this constraint, we proposed that an NSA may decide to focus on subverting the Random Number Generator (RNG) of virtual machines through control of the hypervisor. This has several attractive qualities. First, it is possible to achieve control of the RNG without having to modify the virtual machine (VM) itself, meaning a cloud tenant, even one that inspects checksums of critical system components such as the kernel, should be unable to detect any subversion without specifically searching for artifacts of the used technique. Second, with control over the RNG, cryptographic keys become predictable, and so there is no need to exfiltrate encryption keys out of the cloud infrastructure. Encrypted communications can be monitored from outside the cloud infrastructure with no suspicious key leakage shadowing each new encrypted message. Third, if an NSA can control the RNG with high enough precision, it should be possible to cause the output random numbers to still appear truly random, and only predictable to the NSA, thus maintaining the tenant's security against all non-NSA actors.

In this project we make the following contributions:

- We have created a working attack prototype against the Linux 4.4.6 kernel. We subvert any reads from `/dev/random` and `/dev/urandom` and modify their outputs to a deterministically pseudorandom byte stream. Our prototype does this without modification to the VM using LibVMI[14], a Virtual Machine Introspection (VMI) tool.

- We have implemented a similar, perhaps more impactful, attack against the user space OpenSSL [18] RNG, commonly used by modern software frameworks such as the Apache2 [20] and NGINX [17] web servers. With this attack an NSA can completely subvert the cryptographic key generation on web servers of cloud tenants in a cloud service provider.

- We provide insight into the defense space, including modeling detection mechanisms and how they might be used by cloud tenants to discourage NSAs from subverting the RNGs of their VMs. We successfully implemented a simple detection against the kernel attack we describe in this paper, and further explore how similar defense mechanisms might be used to thwart more sophisticated attacks.

2 Related Works

Virtual machine introspection is a technology that has existed for several years, and that was born out of security-based concerns. In 2003, Garfinkel and Rosenblum developed what is regarded as the first VMI system as a mechanism for isolating an Intrusion Detection System (IDS) from the operating system (OS) of the machine it was protecting [11]. With the development of robust tool sets to perform VMI [9, 14], it might be suspected that VMI is currently being leveraged by public cloud service providers to enhance security guarantees of their systems, a hypothesis which has not yet been confirmed or denied.

With the evolution of VMI and its use in virtual environments, Wang et al. have demonstrated it is possible for guest VMs to detect when they are being introspected on [2] and leverage this knowledge to perform malicious activities between introspection intervals. This is relevant to our own work, as detection of introspection is one of the main defense mechanisms against the attacks we present.

To the best of our knowledge, using the hypervisor to poison a random number generator has only previously been explored by Alt et al. [1]. In this work, the authors modified the QEMU[19] emulator[1] to intercept execution of specific RNG functions in a VM and poison the result. However, this approach may be brittle since it requires a rewrite of QEMU itself to adapt to different kernels or new attack methods. In contrast, our work uses a flexible introspection layer, and can upgrade and reattach to a running VM. Alt et al.'s work also consisted of exploring only the attack surface, leaving defense mechanisms and user space attacks to future work.

[1]Which works in conjunction with the KVM and Xen hypervisors

3 Methodology

3.1 Architecture

In order to explore how VMI might be leveraged to subvert kernel space random number generation, we built a system to simulate a virtual cloud environment. We configured a server (Intel Core i5-650, 3GB RAM) to run the Xen hypervisor [21], with Debian Testing 4.4.6 as the Domain 0 (Dom0, the host) VM and Debian Testing 4.4.6 as the first Domain U (DomU, the Test VM) VM. The architecture of the simulated virtual cloud environment is detailed in Figure 1.

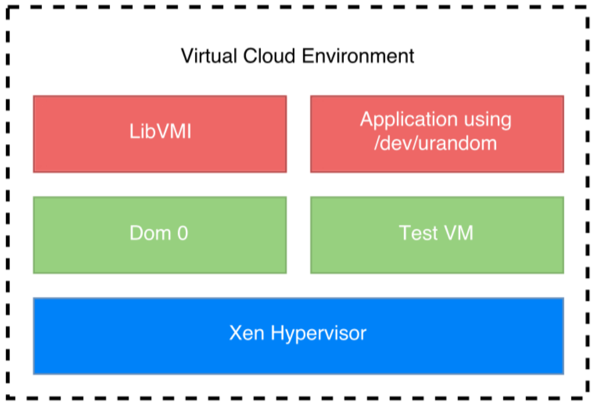


Figure 1: Virtual cloud environment architecture used to deploy RNG subversion attack

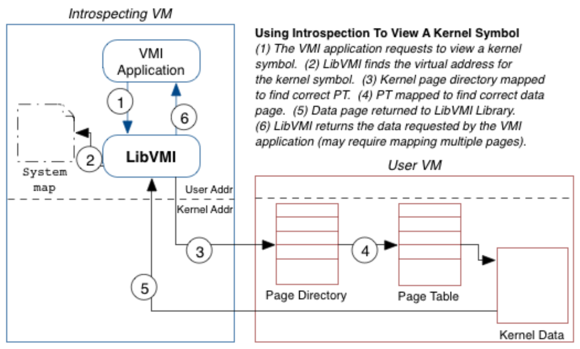
3.2 Using LibVMI

To introspect on the VM, we used the LibVMI library [14], shown in Figure 2, over the GDB-based approach or QEMU-modification performed by Alt et al. [1]. LibVMI interfaces with the Xen hypervisor and provides convenience functions for:

- accessing memory

- resolving kernel symbols to memory addresses

- receiving callbacks when particular events occur



Using Introspection To View A Kernel Symbol
(1) The VMI application requests to view a kernel symbol. (2) LibVMI finds the virtual address for the kernel symbol. (3) Kernel page directory mapped to find correct PT. (4) PT mapped to find correct data page. (5) Data page returned to LibVMI Library. (6) LibVMI returns the data requested by the VMI application (may require mapping multiple pages).

Figure 2: LibVMI memory access model. Our code would be the VMI Application in this diagram. (source: libvmi.com [14])

LibVMI is very powerful, but is also under current development by a small team, and we ran into several cases where features we desired were not yet implemented.

In particular, LibVMI's event monitoring framework is attractive for this sort of attack because it enabled us to register callback handlers and invoke them when particular events occurred. LibVMI supports four types of events: Memory, Register, Interrupt, and SingleStep. Memory events trigger when a particular memory region is read, written to, or executed. Register events trigger when specific control registers are written to. Interrupt events trigger when an interrupt occurs in the monitored VM. SingleStep events trigger after the current instruction executes.

Originally, we intended to register a memory event for when a particular kernel instruction was executed, then pause execution and perform our RNG tampering. Unfortunately, LibVMI cannot currently handle such fine-grained memory accesses[2], and threw an error when we attempted to register the event. In order to get the VM to pause execution at the precise code location we wanted, we devised an alternate strategy.

Instead, when our attack first starts, we pause the VM, regardless of what instruction is currently being executed. We find the memory location of our target instruction, and overwrite it with an Int3 instruction (`0xcc`), saving the original byte to a local variable for later recovery. When executed, the Int3 instruction causes an interrupt, and LibVMI's Interrupt event framework can register a handler for such interrupts. When we receive a callback for the interrupt, we perform whatever introspection necessary, replace the overwritten byte with its original value, thereby restoring the original execution of the kernel code, and register a single-step callback before allowing the VM to resume. Immediately after the next instruction, the single-step callback triggers and we again replace the target instruction's first byte with an Int3 instruction so that on the next iteration we will be able to introspect again. This sequence of events is actually what happens when a debugger such as GDB sets a breakpoint. In our code, we generalized this sequence of actions into a breakpoint-creation function, and we hope to ultimately contribute this functionality upstream to the LibVMI project.

[2]We believe this is a limitation of the current LibVMI/Xen interface and may be supported in the future. LibVMI can only take advantage of runtime information Xen provides, and finer-grained memory monitoring without Xen support would incur orders-of-magnitude more overhead.

3.3 Attacks

3.3.1 Possible Approaches

As in Alt et al.'s work [1], we hypothesized three possible approaches for subverting the Linux RNG from the hypervisor. The first approach was to simply wait for the Linux RNG to generate random numbers, then immediately replace them with deterministic bytes before they could be used by any user application (shown as A in Figure 3). The second approach was to intercept and emulate the *rdrand* x86 instruction, an instruction that asks the CPU to return a random number generated from the hardware RNG[15]. The third approach involved intercepting all input into the Kernel's entropy pool so that its output would become deterministic and predictable (shown as B in Figure 3). Also as in Alt et al.'s work, we determined that the most straightforward solution would be the first option.

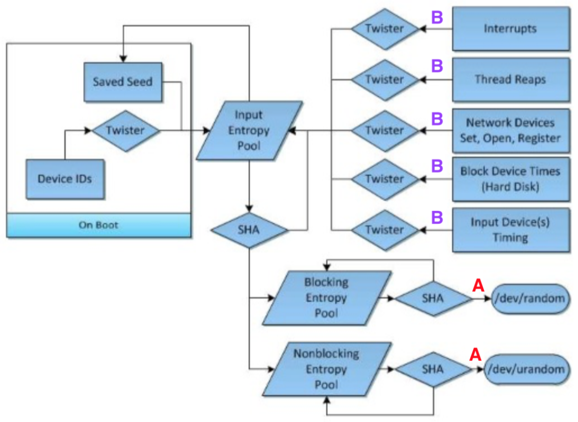


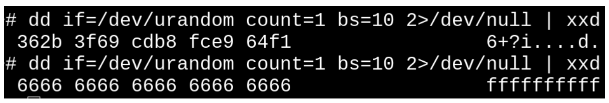
Figure 3: Linux entropy pool management architecture (source: Alt et al. [1])

After discovering that OpenSSL, the cryptographic library used by the popular web servers Apache2 and NGINX, maintains its own entropy pool and RNG in user space, we extended our attack to target the OpenSSL

RNG. We have verified that our attack successfully subverts the `TLS_DHE_RSA_WITH_AES_128_CBC_SHA` Diffie-Hellman key exchange private key on an Apache2 server running on our virtualization platform. Using this attack an NSA can predict the ephemeral shared session key for each HTTPS connection between users on the Internet and service providers using that cloud provider, and thus be capable of real-time Man-in-the-Middle (MitM) attacks against large portions of the Internet.

3.3.2 Kernel Space RNG

Our goal with this attack was to cause any read from the `/dev/random` and `/dev/urandom` file objects to return a predictable sequence of bytes. As our reference test case, we used the *dd* utility to read a configurable number of bytes from these file objects from user space. This is analogous to how other user space applications open and read bytes from these files. Normally, these reads return random bytes, but when our attack executes, any following read will return our signal value, a stream of '`0x66`' bytes (corresponding to the '`f`' character). We also wrote a variant of the attack that emits an attacker-controlled pseudorandom sequence of bytes produced by the unseeded C *rand* function, thereby reducing the expected complexity of the cryptanalysis necessary to predict any generated keys.[3]



```
# dd if=/dev/urandom count=1 bs=10 2>/dev/null | xxd
 362b 3f69 cdb8 fce9 64f1                  6+?i....d.
# dd if=/dev/urandom count=1 bs=10 2>/dev/null | xxd
 6666 6666 6666 6666 6666                  ffffffffff
```

Figure 4: Generating random bytes with and without the hypervisor attack.

As shown in Figure 3 (A), this attack required finding the point in the Kernel's RNG at which random bytes were generated but had not been used. We reverse-engineered the Kernel's RNG by reading through the RNG source code, `random.c`. We found that the appropriate place to introspect was in the *extract_entropy_user* function, just before and after calls to *extract_buf*, which retrieves increments of 10 random bytes from the appropriate entropy pool and returns them in a temporary buffer `tmp`.[4] Examining the compiled code of `random.o`, we discovered that the register holding the address to `tmp` was clobbered after returning from *extract_buf*, so we created two breakpoints: one before and one after the call to *extract_buf*. Before the call, we retrieved the memory address of `tmp` from the `RSI` register, and afterwards, we overwrote 10 bytes at that address with our signal value. It is important[5] to note that only two lines after the call to *extract_buf*, `tmp` is copied into the virtual memory of the user space process that invoked it and is likely much more difficult to access.

3.3.3 User Space RNG

After we completed our attack against the Linux kernel's RNG, we were annoyed to discover that the first non-trivial test we performed to assess the validity of our attack didn't work because the application didn't use the kernel RNG.[6] Specifically, we configured the Test VM to run an Apache2 web server with TLS enabled supporting only the `TLS_DHE_RSA_WITH_AES_128_CBC_SHA`[7] cipher enabled.[8] Viewing the Server Key Exchange packet in Wireshark yields the public values for `p`, `g`, and `pubkey`. `p` and `g` are loaded when the webserver process starts and are static for all connections, but the `pubkey` is generated by taking `g` to the `privkey` power modulo `p`. Unfortunately, `pubkey` was not being generated by taking `g` to the power of our signal value, 256 bytes of `0x66`, modulo `p`, and so we knew that the webserver was not directly using values from `/dev/random` or `/dev/urandom`.

After investigating Apache2's `mod_ssl` module and its corresponding use of the OpenSSL library, we discovered that OpenSSL maintains its own RNG and entropy pool, for which merely one input is `/dev/urandom`. Therefore, we reverse-engineered the mechanism by which OpenSSL generates its random numbers. OpenSSL's `BIGNUM` library contains a function *BN_rand*, which calls *bnrand*,[9] which extracts bytes from the OpenSSL RNG and puts them in a buffer before converting them to a `BIGNUM` object via the *BN_bin2bn* function. These `BIGNUM` objects are then used as parameters for cryptographic operations throughout OpenSSL and Apache2. Exactly like in our kernel attack, we set a breakpoint after the random bytes are generated, immediately before the call to *BN_bin2bn*, and overwrite the random bytes with our signal value.

[3]Future work is needed to evaluate mechanisms for the attacker to use cryptography to increase resistance to fourth-party RNG predictability while maintaining third-party predictability.

[4]By hooking in this one location, we were able to modify the bytes returned by both `/dev/random` and `/dev/urandom`.

[5]For performance reasons. See Section 4.

[6]Other applications, such as Gnu Privacy Guard, *do* use the kernel RNG directly.

[7]The `TLS_DHE_RSA_WITH_AES_128_CBC_SHA` cipher is supported by modern versions of both Chrome (49.0.2623.87) and Firefox (45.0.1), and is representative of currently-used cipher suites. We chose this specific cipher because, unlike elliptic curve versions of the Diffie-Hellman key exchange, it is easy to visually inspect and verify the success of our attack using Wireshark.

[8]Despite statically configuring a select cipher for our tests, we believe our attack works across all forward-secret cipher suites since they all rely on random number generation.

[9]Only the symbol for *BN_rand* is exported, and the assembly code for *BN_rand* makes an unconditional jump into *bnrand*, which is loaded at a dynamic offset.

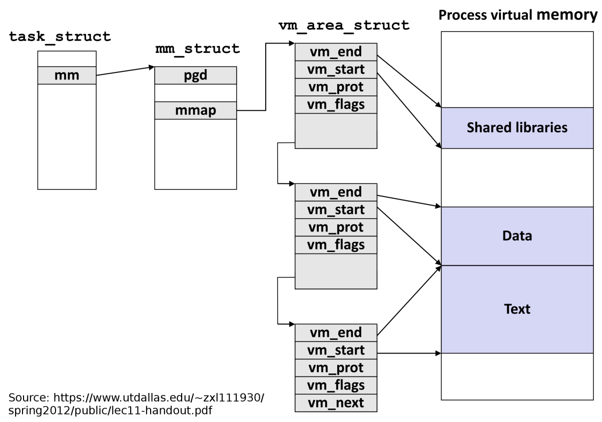


Figure 5: Kernel data structures that it is necessary to walk in order to find appropriate memory offsets in running user space applications (source: Zhiquiang Lin [8])

However, unlike the kernel attack where we could use LibVMI's *vmi_translate_ksym2v* function to translate kernel symbols to virtual memory addresses, user space processes don't maintain a global symbol table. Instead, when a process loads its libraries into memory, the symbols are stored with the library. Each time Apache2 is restarted, the linker will dynamically map each of its libraries into virtual memory at new base memory addresses. Offline, we can identify the memory offsets of the target functions. For instance, *BN_rand* was located at an offset of `0xd5a50` into `libcrypto.so.1.0.2`. To find the base memory address for `libcrypto.so.1.0.2`, we needed to walk the kernel's process list, and for the target process, walk its memory map list for the target library. This data structure is shown in Figure 5.

3.4 Detection

Our detection scheme targets the attack vector where an attacker traps and overwrites random bytes with deterministic bytes before they are copied to user space. Our implementation of this scheme detects attacks on the Linux RNG, but the design of our scheme can be extended to work on a user space RNG or another OS, such as Windows.

In our detection model we assume that an attacker will have an existing attack and when a new VM is launched, will need to make a decision to apply the existing attack, or not. The attacker has limited time to make this decision and so cannot manually inspect the VM's code. We also assume the attacker will not have access to source code of any custom modules the tenant has produced. Instead, the attacker has to rely on automation to find the appropriate attack offsets. Under this model, we believe it is possible to create a module to detect or resist RNG subversion.

We have built a simple prototype to attempt to detect RNG subversion. In general, the RNG can create and manage shadow copies of information that is critical to the operation of the RNG, and validate that the resulting random bytes are valid given the shadow data. Our prototype creates a circular buffer which stores a copy of all outputs from the RNG immediately after they are generated, and hopefully before they are overwritten by the attacker. In an extension of this scheme, we describe modifications which involve creating shadow data-structures for all of the RNG's internal state.

3.4.1 Kernel Detection

In order to detect tampering of the kernel's RNG after bytes have left the blocking or non-blocking entropy, we customize `random.c` to maintain a circular buffer of the most recent 4,000 bytes that have been read from either of its blocking or non-blocking entropy pools[10]. On each call to *extract_buf*, we lock and calculate the return bytes twice. The first calculation is put into the same buffer as the unmodified RNG, `tmp`; the second is placed into our circular buffer. Note that performing this computation within `extract_buf` means that the circular buffer will contain bytes read from all readers in the system of both the blocking and non-blocking entropy pools. When our circular buffer reaches its max size, we dump the contents to the console using `printk`.

The actual detection is done by a user space program which does the following: It periodically reads blocks of ten bytes from /dev/urandom and stores their value in memory until the next buffer dump. Then, the process makes sure that the block is located in the buffer dump.[11] If the block is not found, we infer that the RNG has been subverted and raise an alert to the user.

If the user space application was subverted and is storing a changed block, there is a very minor chance that the block could have been naturally randomly generated. Our circular buffer stores 4000 bytes, or 400 blocks. Each block has 2^{80} possibilities. Thus, there is approximately a $400/2^{80}$ chance that the changed block occurs naturally in the buffer and a false negative would occur. We consider this small enough to ignore. On the other hand, we expect false positives to never occur at all. If the block was randomly generated and was not subverted, it should always be in the shadow buffer.[12]

[10]We chose 4,000 arbitrarily. The only requirement in our implementation is that the size be divisible by ten, because the kernel pulls ten bytes out of the entropy pool at a time

[11]A more efficient implementation would provide an `ioctl` to check the circular buffer.

[12]Discounting possibilities for cosmic rays causing bit flips.

3.4.2 Extension of Detection Scheme

Our detection implementation will only work on attacks which tamper with the RNG output by hooking outside of the *extract_buf* function. Conceptually, we can extend the detection scheme to detect any tampering that occurs inside the RNG by augmenting the RNG to manage shadow instances of each of all of it's data-structures rather than just tracking outputs. For each operation that occurs on a data-structure in the RNG, the RNG completes the same operation the corresponding shadow data-structure. If at any point the shadow data-structures and the original data-structures diverge, the RNG would detect that an outside actor has tampered with the RNG. This detection scheme requires duplicating nearly all computation throughout the RNG, which will ultimately will result in at least 100% overhead to the RNG.

Unfortunately, as in any cat and mouse game, if the attacker is prepared for such a detection method, she can overwrite the shadow data structures as well. We discuss alternate detection approaches in Section 4.

4 Evaluation

Conceptually, it is well understood that the hypervisor, as an interface between the VM and hardware, has complete control over any computation the VM may perform. Therefore, it is no surprise that we can subvert the VM's RNG operation, both in kernel and user memory. However, since our threat model is predicated on the attack's usefulness extending only so far in as it remains stealthy, our evaluation of its success depends on the artifacts it produces and how visible they may be to the tenant operating the VM.

The first, and perhaps most noticeable of these artifacts is purely performance-based. Introspection takes time. Wang et al. [2] detect VMI through the timing delay between when the VM is paused so introspection can occur and when the VM is resumed. In their paper, they found that a timing threshold of 5ms was sufficient to detect introspection on their system. From this, we suspect that any delay of a similar order of magnitude would be consistently detectable.

In our kernel attack, we timed the generation of different size blocks of random bytes using the command

```
time dd if=/dev/urandom count=1 bs=N
```

where N is the number of bytes. The results are shown in Figure 6.

From these results, we can see a clear linear increase in overhead corresponding to the number of bytes read. Our attack incurs approximately 3ms of overhead per 100 random bytes. Breaking this down, 100 bytes corresponds to overwriting ten 10-byte buffers, each of which

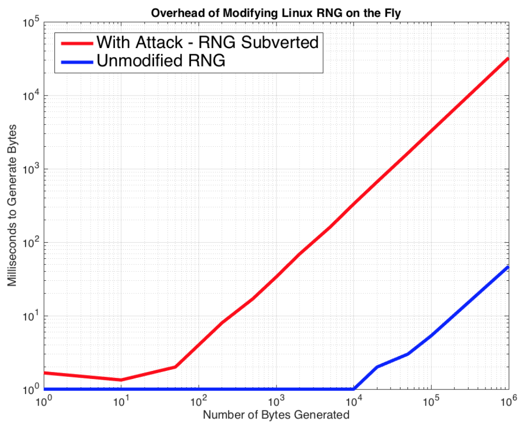


Figure 6: Overhead of kernel attack

requires 2 breakpoints, each of which requires 2 traps to the hypervisor[13]. If we assume that overwriting 10 bytes of memory takes negligible time in comparison with handling an interrupt, we can see that 40 traps to the hypervisor incurs approximately 3ms of overhead. Thus, we believe our current kernel attack is clearly detectable by monitoring the RNG delay.

However, we believe we could significantly reduce this overhead with a more advanced attack. Instead of overwriting each 10-byte buffer of random bytes, if we can successfully trace the copy from kernel memory to user memory, we should be able to overwrite the entire N-byte buffer in a single write operation. However, to do so while maintaining state for an arbitrary number of simultaneous reads may require additional breakpoints for monitoring which processes request random bytes. We identify this as an avenue for improvement that merits more research.

The second artifact we have identified is that both our kernel and user space attacks are relatively brittle. Both are capable of finding the necessary instructions to introspect on in a dynamic manner across reboots, but both also rely on fixed offsets from reference addresses to set the proper breakpoints. This means that, using this technique, a given attack must be manually tailored to each particular kernel or application version to work properly. Since the attacker has control of the hypervisor, they can simply inspect the kernel version before launching the attack, and launch one of many stockpiled attacks against a known version. If an incompatible or unknown kernel is detected, the attacker can always do nothing and simply fail to make the RNG deterministic. We show this decision tree in Figure 7. However, this opens up the possibility of again, detecting a difference in performance between a stock kernel (or application), which is

[13] One to break and introspect, the other to reset the breakpoint.

introspected upon, and a custom kernel (or application), which is not. We identify this as another area in need of more research, both detecting the absence of introspection on a custom kernel, as well as creating more advanced methods of automatically finding appropriate breakpoints in a customized kernel, and what the limits of such an approach may be.

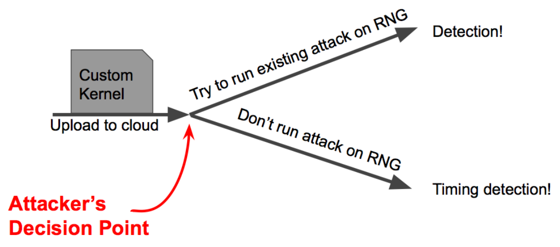


Figure 7: Overhead of RNG Detection

Unlike our attack, in which performance characteristics are critical to its efficacy, the performance of our prototype detection method is immaterial. A kernel specifically instrumented to detect RNG subversion needs not be used in production. Instead, we can use such a detection scheme as a canary to detect if a cloud service provider has been compromised.

In general, an attacker can not detect the semantic behavior of an arbitrary binary. It should be possible to force an attacker to refrain from hooking an RNG and ultimate leak timing information. This allows us to substantially reduce the overhead of our detection scheme, as we will not have to add computation to the RNG but instead merely obfuscate the semantic behavior.

5 Future Work

5.1 Undetectable Predictable Randomness

One obvious detection scheme for an attack on an RNG is to detect a lack of entropy coming from the RNG rather than instrumenting the kernel to detect tampering itself. For example, if our attack generates a stream of `0x66` bytes, it is trivial to detect. If the output is from the C *rand* function, it becomes more difficult to detect by eyeball, but can likely still be statistically detected. An advanced attack could potentially use public-key cryptography to generate a random-looking stream of bytes that could only be predicted with possession of the corresponding private key. However, such an attack, reliant on public key cryptography to generate new bytes, would likely incur additional overhead that may be detectable. We identify this as an area needing further investigation into the tradeoff space of security vs. efficiency while maintaining predictability.

5.2 Limits of VMI Overhead

In our prototype attack, we ran into an unfortunate situation in which our attack introduces significant overhead into the generation of random numbers. We believe that in its current state, this overhead is easily detectable using techniques similar to those used by Wang et al. However, it seems likely that this overhead could be significantly reduced by overwriting the entire N-byte buffer of random bytes instead of `tmp`, the 10-byte temporary buffer.

The first complication to this is that, before `tmp` is copied into the larger buffer, it is copied into the virtual memory of a user space process. As shown in Section 3.3.3, accessing the memory of user space processes is complex, but not necessarily problematic. A much bigger potential issue is that processes spawn and terminate over time, and any process could request random bytes at any time. Thus, unlike our attack against the OpenSSL library that specifically targeted running Apache2 processes, to find the user space virtual memory address of the eventual buffer may require an expensive lookup operation across the kernel process list. This may introduce even more overhead than our original naive attack, or it may be possible to cache common memory locations and optimize lookups over multiple invocations to create an attack with much less overhead. More work is necessary to explore the possible minimal overhead of such an attack.

5.3 Real-world Study

Finally, this entire work is predicated on the suspicion that an NSA may be using VMI to perform a specific category of attack. In our literature review, we were able to find many papers[7, 10, 4] on VMI, but no evidence that any *public* cloud service provider introspects on their tenant's VMs in any way. We were also unable to find any cloud provider that offers introspection as a service, which we thought may be a valid business model since VMI is often used for intrusion detection. We therefore identify a need for a more comprehensive survey and measurement study that answers the question of, ”Who is using VMI, and why?”

6 Conclusion

In this project we investigate how an NSA can leverage its resources and abilities to gain widespread access to modern communication and web services while avoiding public criticism through security and stealth. We focus on the scenario in which an NSA has superuser privileges on a subset of the host machines of a cloud service provider. Specifically, we investigate how an NSA can subvert a virtual machine's Random Number Generator to produce deterministic outputs and present two prototype attacks that demonstrate its feasibility.

Our first attack subverts read operations from /dev/random and /dev/urandom and allows an attacker to produce a deterministic psudeorandom byte stream. Our extension of this attack works against user space RNGs, specifically the OpenSSl RNG which is used by modern web servers such as Apache2 and NGINX. These attacks create artifacts that likely enable detection by a cloud tenant. We present an initial detection scheme and discussion of the issues for detecting this category of attacks performed by an advanced adversary.

This paper is hopefully the first of many that explore NSA-level attacks using the hypervisor. Our work in this paper investigates a particular kind of attack against an RNG wherein the random bytes are simply overwritten before they can be used. However, there are many other avenues for similar attacks, and future work is needed to determine the feasibility of these other attacks, as well as their real-world feasibility.

References

[1] 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.

[2] Gary Wang et al. "Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring". In: *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.: USENIX Association, Aug. 2015. URL: https://www.usenix.org/conference/woot15/workshop-program/presentation/wang.

[3] April Glaser. *After NSA Backdoors, Security Experts Leave RSA for a Conference They Can Trust*. Jan. 30, 2014. URL: https://www.eff.org/deeplinks/2014/01/after-nsa-backdoors-security-experts-leave-rsa-conference-they-can-trust.

[4] Adrian L. Shaw et al. "Forensic virtual machines: dynamic defence in the cloud via introspection". In: *IEEE International Conference on Cloud Engineering (IC2E)*. 2014. URL: http://sacko.uk/pdf/2014.1.pdf.

[5] Olga Khazan. *The Creepy, Long-Standing Practice of Undersea Cable Tapping*. July 13, 2013. URL: http://www.theatlantic.com/international/archive/2013/07/the-creepy-long-standing-practice-of-undersea-cable-tapping/277855/.

[6] Mandient. *APT1. Exposing One of China's Cyber Espionage Units*. Feb. 19, 2013. URL: http://intelreport.mandiant.com/Mandiant_APT1_Report.pdf.

[7] Chris Benninger et al. "Maitland: Lighter-weight VM introspection to support cyber-security in the cloud." In: *IEEE 5th International Conference on Cloud Computing (CLOUD)*. 2012. URL: http://christophermatthews.ca/files/bare_conf.pdf.

[8] Zhiqiang Lin. *CS 6V81-05: System Security and Malicious Code Analysis, Understanding the Implementation of Virtual Memory*. Feb. 29, 2012. URL: https://www.utdallas.edu/~zxl111930/spring2012/public/lec11-handout.pdf.

[9] Bryan D Payne. "Simplifying virtual machine introspection using libvmi". In: *Sandia report* (2012).

[10] A.S. Ibrahim et al. "CloudSec: a security monitoring appliance for Virtual Machines in the IaaS cloud model". In: *5th International Conference on Network and System Security (NSS)*. 2011. URL: http://researchbank.swinburne.edu.au/vital/access/services/Download/swin:23719/SOURCE2.

[11] Tal Garfinkel, Mendel Rosenblum, et al. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." In: *NDSS*. Vol. 3. 2003, pp. 191–206.

[12] *Amazon Web Services*. URL: https://aws.amazon.com/.

[13] *Google Cloud Compute*. URL: https://cloud.google.com/.

[14] *LibVMI: Virtual Machine Introspection*. URL: http://libvmi.com/.

[15] John M. *Intel® Digital Random Number Generator (DRNG) Software Implementation Guide*. URL: https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide.

[16] *Microsoft Azure*. URL: https://azure.microsoft.com/en-us/.

[17] *NGINX*. URL: https://www.nginx.com/.

[18] *OpenSSL Project*. URL: http://www.openssl.org/.

[19] *QEMU: Open Source Processor Emulator*. URL: http://wiki.qemu.org/Main_Page.

[20] *The Apache HTTP Server Project*. URL: `https://httpd.apache.org/`.

[21] *The Xen Project*. URL: `http://www.xenproject.org/`.