Designing a minimal operating system to emulate 32/64bits x86 code snippets, shellcode or malware in Bochs

Elias Bachaalany, @allthingsida

originally presented at REcon 2012

Overview

- Introduction
- System overview
- System design
- Demo

Introduction

What's this talk about?

- How to design a minimal operating system for the purpose of debugging code snippets or malware
- Challenges faced

Motivation

- Static analysis is great, but not all the time:
  - encrypted/packed/obfuscated
  - long/complex algorithms
- Debugging shellcode
- Debugging a selected piece of code or subroutine
- Emulate an MS Windows malware from a non MS Windows environment
- Emulation should be as accurate as the real processor

Why use emulators and VMs?

- Provides an environment for quick and easy experimentation
- Run code without risk of infection
- Dynamic code analysis
  - Unpacking
  - Algorithm recovery
    - Crypto algorithms
    - Hashing algorithms
    - etc...
- Security research

Candidate emulator

To debug malware or arbitrary x86/x64 code snippets, we need a programmable emulator with this minimal functionality:

- Emulation control:
  - start, stop, suspend
  - manage disk images
- Debug control:
  - single stepping, tracing
  - register manipulation
  - breakpoints: add, delete, disable
  - physical memory read/write, ...

Reinventing the wheel?

There are plenty of emulators/VM, why not choose an existing solution?

- VM products:
  - VMware, VirtualPC, QEmu, Bochs, ...
- Emulation libraries:
  - pyemu, x86emu, ida-x86emu, libemu, ...

Selecting an emulator (1)

While emulation libraries are highly programmable and simple to use, they:

- are not necessarily mature enough: wrong instruction emulation in some cases
- do not support all instructions: easily defeated if obscure (or unsupported) instructions are used
- emulation tend be slower than inside a VM

Selecting an emulator (2)

On the other hand, popular VM products are:

- mature: very accurate emulation
- fast: they employ dynamic binary translation or hardware aided virtualization
- programmable: emulation state and debug control are provided
  - VMWare can be controlled with a gdb stub
  - Bochs provides a plugin system or a command line debugger (bochsdbg.exe)
  - Etc…
- capable of full OS emulation: debug a complete operating system and thus they support obscure instructions

System overview

System overview

This emulation system is composed of:

- A programmable emulator: Bochs, Qemu, Vmware, …
- Driver program (also referred to as host)
  - Prepares the disk image
  - Provides the minimal operating system
  - Communicates with the emulator
- Code to emulate
  - Packed malware
  - Shellcode
  - Code snippets

System overview

Input files
- .dll | .so file
- PE | ELF file
- Binary | Shellcode

Image creation
- PE | ELF loader
- Processor setup
- Physical memory content setup

Emulator
- Bochs, VMWare, Qemu, etc…
- Debug control API
  - Single stepping
  - Bpt managment

Image execution engine
- MBR
- Kernel
- API and OS emulation
- Host / Guest communication
- setup

System overview

Image creation

- Setup the processor structures
  - Set up protected mode
  - GDT / IDT / PTEs
- Transform the code to be emulated into a disk image
  - File loaders (PE loader, ELF loader, etc…)

Image execution

- Boot the emulator
- Handle system services
- Guest<->Host communication
- Target OS emulation (exception handling, system structure emulation, etc…)

Disk image creation

Disk image creation

- Image creation
  - Image file loader
    - Structured input: PE, ELF, …
    - Shellcode
  - OS structure preparation
    - Page directory setup
    - Physical memory contents
  - OS file system design
    - VM image file format
      - Represent all needed input in a single disk image

The virtual memory manager

- The driver program implements a virtual memory manager class:
  - The virtual memory is set up prior to execution
  - Page table entries setup is based on the input file(s)
  - For example PE file loader will dictate the VM layout
  - VM libraries are written in C++
  - Each VMM operation has side effects on the physical memory:
    - Allocate virtual memory -> setup proper PTEs
    - …
  - All virtual memory operation side effects are **serialized** and flushed to the disk image

The virtual memory manager

- The VMM class implements methods such as:

```
// Apply page table entry default attributes
// attr has one of the PGATTR_xxx constants
virtual void apply_attr(vmm_page_attr_t attr) = 0;

// Maps multiple pages
virtual vmm_err_t map_many_pages(
  uint16 selector,
  ea_t offs,
  ea_t *start_phys_loc,
  size_t sz,
  bool skip_already_mapped,
  bool user_phys_loc) = 0;

// Maps a single page
virtual vmm_err_t map_page(
  uint16 selector,
  ea_t offs,
  ea_t *phys_location,
  bool user_phys) = 0;
```

The virtual memory manager

- When emulating x86, the x86_vmm class implements the map_page() so it creates the appropriate PDEs and PTEs

```
vmm_err_t internal_x86_vmm_t::map_page_ex(
  uint16 selector,
  uint32 offs,
  uint32 *phys_location,
  uint32 *ptr_pde,
  uint32 *ptr_pte,
  page_dir_entry_4kb_t *o_pde,
  page_table_entry_t *o_pte,
  bool user_phys)
```

- The VM class simply serializes what is needed to be written to the physical memory when a map_page() is requested
- All memory transactions are **recorded** into the serializer.

The virtual memory manager

- The VMM operation serializer can be subclassed so it flushes the side effects to a file (in the case of disk image creation) or to the virtual machine (during the image execution phase for example).
- ```
  class vmm_serializer_t
  {
  public:
    virtual bool serialize(
        const uint64 addr,
        const void *buffer,
        const size_t sz) = 0;
    virtual ~vmm_serializer_t() { }
  };
  ```
  ```

The virtual memory manager

- Here we can see how a change page attributes operation simply serializes (or records) what changes are needed to be applied to the physical memory

```
vmm_err_t internal_x86_vmm_t::ch_page_attr(
    uint16 selector,
    uint32 offs,
    vmm_page_attr_t attr)
{
  page_table_entry_t *pte;
  uint32 ptr_pte;

  if ( !ch_page_attr_ex(selector, offs, &ptr_pte, &pte, attr) )
    return vmm_err_not_mapped;

  serialize(ptr_pte, pte, sizeof(*pte));

  return vmm_err_ok;
}
```

The virtual memory manager

- Since the emulation system need to support x64 bits, we had to implement an x64 memory manager class

```
vmm_err_t map_page_ex(
  uint64 linear,
  uint64 *outphys,
  uint64 *phys = NULL,
  bool remap = false,
  pte_4kb_64_t *pte_attr = NULL);

vmm_err_t map_many_pages_ex(
  uint64 linear,
  size_t sz,
  vmm64_mdl_t *mdl,
  bool remap = false,
  uint64 *outphys1 = NULL,
  uint64 *phys = NULL,
  pte_4kb_64_t *pte_attr = NULL);
```

- This class supports Page-Map Level 4 (PML4) tables

File loaders

- The emulation system should be able to interpret an executable or raw instruction stream:
  - PE files
  - ELF files
  - Shellcode
  - Code snippets
- A PE loader is implemented to parse PE files:
  - Parse the main executable
  - Parse dependencies
  - Resolve imports
  - ….

PE loader (1)

- The PE loader class:
  - Knows how to parse PE files and their dependencies:
    - Import resolution
    - Relocation handling
    - Proper handling of forwarded API
  - It needs a virtual memory manager class to map the PE sections to the virtual memory
- Additionally, the PE loader can interpret a meta file so it knows how to deal with dependencies:
  - Can generate dummy DLL stubs
  - Map a DLL as it is
  - Handle API emulation via scripting

PE loader (2)

- The PE loader is also responsible for setting up:
  - The PEB
  - The TIB
  - The NT structures:
    - NT32_RTL_USER_PROCESS_PARAMETER
    - NT32_LDR_MODULE (Load and Init order)
    - …
- The PE loader also knows how to:
  - Module management:
    - LoadLibrary(), GetProcAddress(), etc…
  - VA to Physical conversion (and vice versa)
  - etc…

PE loader – startup configuration (1)

- The PE loader reads a special file that instructs it how to interpret modules and API emulation

- The startup supports such directives:
  - "map-module: path=path_to_module, load_address=[ADDR|ASLR|default]" <- Map the module as it is in the VM
  - "imitate-module: path=path_to_module, load_address=[ADDR|ASLR|default]" <- Generate a dummy stub containing all the exported entries

PE loader – startup configuration (2)

- Continued….:
  - "map-file: va=load_address, file=file.bin,page_prot=flags" <- map a binary file to the desired VA (load shellcode into the emulation environment for instance)
  - "map-mem: va=load_address, size=SZ, page_prot=flags" <- maps uninitialized memory

These directives instruct the PE loader how to load and map PE files and their dependencies

PE loader - modules configuration (1)

- Each module described in the startup configuration file has its own configuration script / file:
  - Implement certain API emulation of the module
  - Redirect certain API:
    - Redirect functionality to another module
    - Redirect functionality to a script
- The module configuration file contains directives such as:
  - "func: name=GetProcAddress, entry=redirModule.NewApi" <- to redirect an API in this module to another module
  - "func: name=FuncName, purge=N, retval=123" <- Generate a dummy API stub that always returns 123 and purges N bytes from the stack

PE loader - modules configuration (2)

- Continued:
  - “func: name=FuncName, entry=ScriptFunctionName” <- to redirect an API in this module to a function in a script file on the host

- For example, “kernel32.py” may contain the following:

```
///func: name=Beep, entry=beep, purge=8
def beep():
  param1 = Emu.GetParam(1)
  param2 = Emu.GetParam(2)

  print "I am Beep(%d, %d)\n" % (param1, param2)

  # The emulated function returns 1:
  SetRegValue(1, "EAX")

  # Return value controls execution of the debugged application:
  #   1 = suspend execution
  #   0 = continue transparently
  return 0
```

PE loader - Dummy API stub

- This dummy stub is generated due to an entry in kernel32.py as:
- "func: name=GetProcessAffinityMask, purge=12, retval=0"
- The stub calls a dummy entry in the kernel <- this makes it easy to break on all dummy (not overwritten API calls)

```
7DD63647 kernel32_GetProcessAffinityMask proc near
7DD63647 mov     eax, offset kernel32_GetProcessAffinityMask
7DD6364C call    near ptr bochsys_BxUndefinedApiCall
7DD63651 mov     eax, 0 ; <- retval
7DD63656 retn    12 ; <- purge value
7DD63656 kernel32_GetProcessAffinityMask endp
7DD63656
```

PE loader - Script API stub

- This stub allows you to implement an API via a script.
- It uses the guest-to-host calls (explained later)
- user32.py may contain:

```
#///func: name=MessageBoxA, entry=messagebox, purge=0x10
def messagebox():
  param2 = Emu.GetParam(2)
  print "[Python] MessageBoxA() has been called: %x %s\n" %
    (param2, Emu.GetSzString(param2))
  SetRegValue(1,"eax")
  # continue execution
  return 0
```

Causing the following stub to be generated:

```
USER32.dll:7DC53532 user32_MessageBoxA proc near
USER32.dll:7DC53532 mov     eax, offset user32_MessageBoxA
USER32.dll:7DC53537 call    near ptr bochsys_HostCall
USER32.dll:7DC5353C retn    10h
USER32.dll:7DC5353C user32 MessageBoxA endp
```

PE loader - Forwarded API stub

- This stub allows you to redirect the functionality of an API to another module / API:

```
#///func: name=GetProcAddress, entry=bochsys.BxGetProcAddress
```

```
KERNEL32.dll:7DD63642 kernel32_GetProcAddress proc near
KERNEL32.dll:7DD63642 jmp     bochsys_BxGetProcAddress
KERNEL32.dll:7DD63642 kernel32_GetProcAddress endp
```

- This stub redirects kernel32!GetProcAddress to the kernel's GetProcAddress()

Shellcode / Code snippet loader

- The shellcode / code snippet loader is very simple:
  - Read the startup configuration file and map the needed binary images into the virtual machine
  - The virtual memory manager is instructed to allocate and map pages per the configuration file

PE loader + VMM + Disk file

This is how the system looks so far:

Loader
- Parse PE file
- Load dependencies
- Etc…

- Alloc mem
- Write bytes

VMM
- Allocate pages
- Serialize VM operation side effects
- etc…

Flush VMM contents to disk

Disk image writer
- Write MBR
- Write the kernel
- Write GDT/IDT
- Flush serialized bytes
- Etc….

Shellcode + VMM + Disk file

This is how the system looks so far:

Loader
- Load binary contents

- Alloc mem
- Write bytes

VMM
- Allocate pages
- Serialize VM operation side effects
- etc…

Flush VMM contents to disk

Disk image writer
- Write MBR
- Write the kernel
- Write GDT/IDT
- Flush serialized bytes
- Etc….

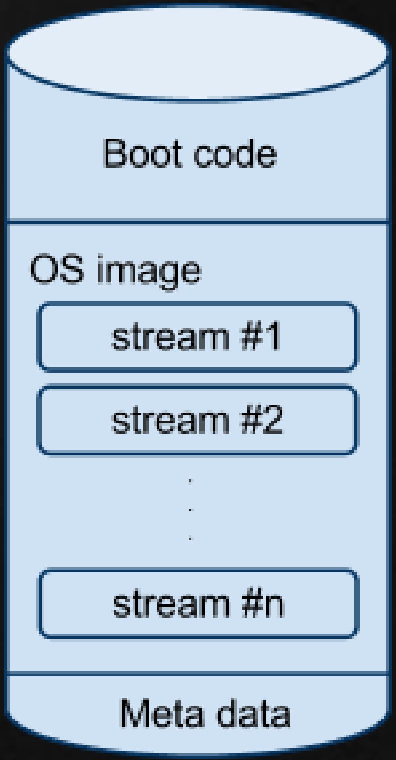
Boot images on Intel compatible CPUs

- On Intel compatible processors, a bootable disk image should have an MBR at the first sector
- The MBR loads a boot sector from the active partition
- The boot sector then loads the kernel and starts the operating system
- In our case, only the MBR is used (two sectors). It will load the kernel and the other components

Disk image format - Overview

The disk image is composed of:

- Boot code (MBR at sector zero)
- The OS image
  - GDT/IDT setup
  - Page directory setup
  - Physical memory contents
- Meta data appended at the end of the disk



Disk image format - MBR

- The MBR occupies two sectors
- How it works and what it does is discussed in the “Image Execution” section

Disk image format – OS image (1)

- The OS image contains everything that was serialized during the input loading time
- Everytime the PE loader maps a PE file or its dependencies in memory, the requests are recorded into the VMM serializer class

Disk image format – OS image (2)

- The OS image simply contains a stream header followed by a list of streams
  - Stream header:
    - number of streams
    - header version
    - etc…
  - One or more streams of the following format:
    - stream_size: the size of the stream
    - stream_attributes: some attributes
    - physical_memory_location_to_load_at: where to write
    - stream_bytes: the bytes to write to physical memory

Disk image format – OS image (3)

- The driver program creates streams indirectly each time a memory is allocated through the VMM class
- The driver uses the VMM to allocate / setup the IDT and GDT contents at a fixed / reserved address (same as IDT and GDT addresses in Windows XP)
- The driver will flush the system structures (GDT/IDT) to the disk image into streams

Disk image format – OS image (4)

- Additional meta-data is appended at the end of the disk image
- The meta-data is not part of the mini-OS but is used by the driver:
  - Store cache data
  - Store configuration blob
  - Etc…

Image execution

Image Execution - overview

- The master boot record (MBR)
  - Load the streams
  - Jump to kernel

- The OS kernel
  - Responsible for target OS emulation
    - Exception handling / emulation
    - System structure emulation (PEB, TIB structs…)
    - Etc…
  - Host to guest communication
  - API emulation / extension (through guest-to-host communication)

Boot process

- Enter unreal mode
  - Provide 4GB physical memory access from 16-bit real mode
- Load the streams
  - Verify the stream header
  - Load all streams:
    - GDT/IDT
    - Page directory setup
    - OS image stream: it is part of the streams. Entrypoint is patched-in during the disk image creation phase
    - Other streams
- Switch to protected or long mode
- Transfer execution to the kernel

Boot process

The boot code:

- 16-bit real mode code
- Enters unreal-mode to access memory > 1MB
- Verifies the OS image format
- Load OS image to physical memory
- Page table entries are also loaded
- Load the kernel
- Jump to the kernel entry point

Boot process

```
; Clear bios boot text
call clear_screen

; Load remaining boot sector code
call load_boot_sector

; Show loading message
call display_loading_message

; Enable 4GB access
call setup_unreal

; Load all streams
call load_objs

; Load the kernel
call load_kernel
```

```
; Loads the appropriate kernel
load_kernel:
  mov eax, [data.kernel_flags]
  test eax, KERNEL_FLAG_64BITS_MODE
  jnz short .64
  ; Start 32bit kernel
  call load_32bit_kern
 .64:
 ; Start 64bit kernel
  call load_64bit_kern
```

Memory layout at the kernel start (1)

The memory layout

- First 1MB reserved
- At 8MB the PDBR (CR3)
- Initial page directory setup
- Uninitialized pages:
  - BSS sections of modules
- Initialized pages:
  - Main program memory
  - Dependencies:
    - Modules
    - Injected binary files

0-1MB

- IVT
- MBR
- …

>= 8MB

- CR3 -> PDBR
- PDE / PTEs

> 8MB + size(PTEs)

- Main module
- Dependencies
- BSS memory
- Etc….

Memory layout at the kernel start (2)

Physical memory (0-4GB)

- MBR (identity mapped)
- Main module
- Dependencies
- Etc….

Virtual memory

- MBR (identity mapped)

0x401000 - END

- Main module

0x10000000 – END_1

- Module 1….

0x1A400000– END_2

- Module 2….

0xE0000000 – END OS

- OS kernel
- IDT handlers…

0x8003F000 - 0x80040400

- GDTR -> GDT
- IDTR -> IDT

- The VMM class assures proper page table entry setup prior to execution
- The kernel does not update the PTEs after it runs (it is done with guest-to-host calls instead)

Kernel services

- Setup IDTs (for exception handling)
- Exception dispatcher
- Dispatch TLS callbacks
- Transfer execution to user code
- Handle program termination
  - Exit callbacks:
    - TLS or DLLMain()
    - Call exit script (guest-to-host)
- Guest-to-Host and Host-to-Guest communication
- Emulation environment
- Debugging facilities

Kernel initialization - Overview

- At the time of MBR-to-kernel transfer all memory content is set up already
- The kernel starts in Ring 0
- Ring 0 initialization code:
  - Setup R0 stack space
  - Build and setup IDT, GDT and TSS
  - Setup the Ring3 FS selector
  - Install the unhandled exception handler
  - Init FPU
  - Jump to Ring 3 initialization code in the kernel
    - Switch to Ring3 via an IRET instruction
- Ring 3 initialization code:
  - Parse the input file and decide what to do
    - Dispatch TLS callbacks / DLLMain()
    - Or just call main program's entrypoint
    - -> Return to ExitProcess() after the target main() terminates

Kernel initialization – Interrupts (1)

- The following interrupts are set up with CPL=0
  - DIVIDE_BY_ZERO (0x00): Handles division by zero
  - SINGLE_STEP (0x01): Handles single stepping
  - INVALID_OPCODE (0x06): Handles invalid opcodes exceptions
  - STACK_EXCEPTION (0x0C): Handles stack exceptions
  - GPF (0x0D): Handles general exception faults
  - FLOAT_P_ERROR (0x10): Handles floating point errors

- Those interrupts are called by the emulator (or the CPU) when a fault or exception takes place

Kernel initialization – Interrupts (2)

- The kernel allows certain interrupts to be called from R3 to emulate the desired operating system.

- The following interrupts are set up with CPL=3
  - BREAKPOINT (0x03): Handles breakpoints. R3 instructions should be able to issue an INT3 (0xCC or 0xCD, 0x03) without getting a GPF
  - INTO (0x04): Interrupt on overflow is allowed from R3

Kernel initialization – Interrupts (3)

- All interrupt handlers share the same stub
- The stub stores the registers context into a CONTEXT compatible structure
- Control is then passed from R0 (the interrupt handler) to the R3 exception dispatcher
- The exception dispatcher will convert *raw* exceptions into Windows exceptions

Kernel initialization - Interrupts (4)

- This is how the interrupt handler stubs look like:

```
Int0x00_Handler:
mov      exception_code, 0
jmp      R0InterruptHandler

Int0x01_Handler:
mov      exception_code, 1
jmp      R0InterruptHandler

Int0x03_Handler:
mov      exception_code, 3
jmp      R0InterruptHandler

Int0x06_Handler:
mov      exception_code, 6
jmp      R0InterruptHandler
```

```
Int0x0C_Handler:
mov      exception_code, 0Ch
jmp      R0InterruptHandler

Int0x0D_Handler:
mov      exception_code, 0Dh
pop      exception_errno
jmp      R0InterruptHandler

Int0x0E_Handler:
mov      exception_code, 0Eh
pop      exception_errno
jmp      R0InterruptHandler

Int0x10_Handler:
mov      exception_code, 10h
jmp      R0InterruptHandler

Int0x04_Handler:
mov      exception_code, 4
jmp      R0InterruptHandler
```

Kernel initialization - Interrupts (5)

- Save the registers

```
EXPORT R0InterruptHandler, 0
 .copy_regs:
  ; General registers
  mov dword [_g_raw_excp+raw_exception_context_t.CONTEXT+CONTEXT._Eax], eax
  mov dword [_g_raw_excp+raw_exception_context_t.CONTEXT+CONTEXT._Ebx], ebx
  mov dword [_g_raw_excp+raw_exception_context_t.CONTEXT+CONTEXT._Ecx], ecx
  mov dword [_g_raw_excp+raw_exception_context_t.CONTEXT+CONTEXT._Edx], edx
  mov dword [_g_raw_excp+raw_exception_context_t.CONTEXT+CONTEXT._Esi], esi
  mov dword [_g_raw_excp+raw_exception_context_t.CONTEXT+CONTEXT._Edi], edi
  mov dword [_g_raw_excp+raw_exception_context_t.CONTEXT+CONTEXT._Ebp], ebp

  ; Copy page faulting address
  mov eax, cr2
  mov dword [_g_raw_excp+raw_exception_context_t.page_fault_addr], eax

  ; Copy debug registers
  mov eax, dr0
  mov dword [_g_raw_excp+raw_exception_context_t.CONTEXT+CONTEXT.Dr0], eax
```

- Return to ring3

```
.goto_r3_dispatcher:
 .
 .
 .
 mov dword [esp+0x00], _R3ExceptionDispatcher@4
 .
 .
 iret
```

Kernel initialization – Interrupts (6)

The kernel will convert the raw instructions to Windows exceptions:

```
DWORD WINAPI R3ExceptionDispatcher(
 struct _EXCEPTION_REGISTRATION_RECORD *List)
{
 switch ( exception_code )
 {
  case INTNUM_DIVIDE_BY_ZERO:
   // could also be EXCEPTION_INT_OVERFLOW
   rec.ExceptionCode = EXCEPTION_INT_DIVIDE_BY_ZERO;
   break;
  case INTNUM_INVALID_OPCODE:
   rec.ExceptionCode = EXCEPTION_ILLEGAL_INSTRUCTION;
   break;
  case INTNUM_PAGE_FAULT:
   rec.ExceptionCode = EXCEPTION_ACCESS_VIOLATION;
   rec.NumberParameters = 2;
   // page fault generate a special error code format:
   // bit 3,2,1: (U/S)(R/W)(P)
   rec.ExceptionInformation[0] = (exception_errno & 2) ? 1 : 0;
   rec.ExceptionInformation[1] = page_fault_addr;
   break;
 }
```

Kernel initialization – Interrupts (7)

- Then the kernel will walk the SEH list and call the handlers

```
while (List != (struct _EXCEPTION_REGISTRATION_RECORD *)-1)
 {
  if (List->Handler(&rec, List, &context, NULL) == ExceptionContinueExecution)
  {
    handled = 1;
    break;
  }
  List = List->Prev;
 }

 if (!handled)
   return UnhandledException();

 return R3ExceptionDispatcherReturnToR0();
}
```

- Return back to R0 so we restore context registers and then finally transfer back to user mode (R3)

Kernel initialization – Syscalls

- The kernel allows system calls (from R3 to R0)
- A SYSCALL (0x2E) entry is created in the IDT with CPL=3
- It allows system calls to the kernel from user mode

- A short list of supported system calls
  - R3INVALIDATE_CACHE: Allows the user mode code to call the privileged instruction INVPLG to invalidate the cache ;!

  - R3EXCEPTIONDISPATCHERRETURNTOR0: Allows the R3 exception dispatcher to resume back to R0

- System call service number is passed via the EAX register:

```
mov eax, SYSCALL_NUM
int 0x2E
```

Dispatching TLS callbacks (1)

- TLS callbacks if present are parsed from the PE header
- They are called before the entry point and at the exit of the program
- TLS callbacks are dispatched within a try/except block

Dispatching TLS callbacks (2)

```
void WINAPI DispatchTlsCallbacks(
  LPVOID ImageBase,
  PIMAGE_NT_HEADERS inh,
  PIMAGE_DATA_DIRECTORY tls_dir,
  DWORD dwReason)
{
  // We want to save caller's return address if any exception occurs,
  // then perhaps exception handler wants to return to caller
  g_tls_jump_back.Eip = (DWORD) _ReturnAddress();

  // TLS present?
  if (
    inh->OptionalHeader.NumberOfRvaAndSizes > IMAGE_DIRECTORY_ENTRY_TLS
    &&
    tls_dir->VirtualAddress != 0)
  {
    PIMAGE_TLS_DIRECTORY32 tls =
      (PIMAGE_TLS_DIRECTORY32) ((DWORD)ImageBase + tls_dir->VirtualAddress);

    if (tls->AddressOfCallBacks != 0)
    {
      PIMAGE_TLS_CALLBACK *cb = (PIMAGE_TLS_CALLBACK *)tls->AddressOfCallBacks;
      DWORD i;

      // Walk through TLS callbacks
      for (i=0;cb[i] != NULL;i++)
          cb[i](ImageBase, dwReason, reserved);
    }
  }
}
```

Guest-to-host communication (1)

- API emulation takes place on the host side (outside the VM):
  - API calls are intercepted in the emulator using a control breakpoint
  - The driver inspects the EAX register -> API index
  - Checks if index is registered with a script function
  - Invokes the script code -> can modify the VM registers and memory contents
  - Resume the breakpoint -> resumes VM

Guest-to-host communication (2)

- Example of emulated function stubs:

```
kernel32!Beep:
mov   eax, 7DD6139Ah ; index of k32!Beep
call  bochsys_BxHostCall
ret   8

user32!MessageBoxA:
mov     eax, 7DC53532h
call    bochsys_BxHostCall
retn    10h

bochsys!BxHostCall:
nop
nop ; Control breakpoint here
nop
retn
```

Guest-to-host communication (3)

- Host receives a BP event -> checks the API emulation control breakpoint -> pass to script

```
int can_handle_breakpoint(debugevent_t &ev)
{
  regs_t &regs = ev.regs;

  if ( regs.rip != bp_hostcall.addr )
          return -1; // Just ignore

  // Do we know this address?
  func_ctx_t *ctx = find_func_ctx(regs.rax);
  if ( ctx != NULL && ctx->func_type == FUNCTYPE_FWDSCRIPT )
    return run_script_function(ctx->entry.c_str());
  else
    return -1;
}
```

Guest-to-host: System services (1)

- Some core operating system API are a special case of the guest-to-host communication
- For example, a VirtualAlloc() call will be intercepted by the control breakpoint and then passed to a specialized function:
  - Parse parameters from the VM stack
  - Use the PE / VMM module to allocate memory
  - Serialize PDE/PTE allocations from the VMM class
  - De-serialize the changes back to the VM physical memory
  - Flush the page cache invplg ;!

Guest-to-host: System services (2)

```
// Allocates memory and also updates the emulator's page table
bool mem_alloc_live(
  ulongptr_t &addr,    size_t sz,
  vmm_page_attr_t pg_attr)
{
  vmm_pg_serializer ser;
  vmm_serializer_t *oldser = vmm->set_serializer(&ser);

  sz = align_up(sz, X86_PAGE_SIZE);
  bool ok = vmm->mem_alloc(addr, sz, pg_attr);
  if ( ok )
    ok = upload_serialized_streams_to_emulator(&ser.get_list());

  vmm->set_serializer(oldser);

  return ok;
}
```

Host-to-guest communication

- Host needs to call inside the VM
- This is achieved via ROP like technique:
  - Push the parameters on the stack
  - Save input registers
  - Pass more parameters into the registers
  - Set EIP = Function to be called
  - Set [ESP] = Control BP
  - Resume control -> Call the guest
  - Stop on Control BP
  - Restore registers

Implementations

- This system has been implemented as a debugger plugin for IDA Pro
- The emulator used was Bochs
  - Open source
  - Programmable
- The minimal kernel (or OS) is implemented in C and Assembly
- There are 32bits and 64bits versions of this mini kernel

Practical use

- Shellcode emulation
- Packed PE emulation
- Code snippets emulation
- Emulating 64bit shellcode on 32bit hosts

Demos

- Shellcode emulation
- Packed PE emulation
- Code snippets emulation
- Emulating 64bit shellcode on 32bit hosts