J Comput Virol (2011) 7:83–93
DOI 10.1007/s11416-009-0139-z

ORIGINAL PAPER

Stealth malware analysis from kernel space with Kolumbo

Julien Desfossez · Justine Dieppedale · Gabriel Girard

Received: 5 January 2009 / Accepted: 4 November 2009 / Published online: 9 December 2009

Abstract Most of today's malware are able to detect traditional debuggers and change their behavior whenever somebody tries to analyze them. The analysis of such malware becomes then a much more complex task. In this paper, we present the functionalities provided by the Kolumbo kernel module that can help simplify the analysis of malware. Four functionalities are provided for the analyst: system calls monitoring, virtual memory contents dumping, pseudo-breakpoints insertion and eluding anti-debugging protections based on `ptrace`. The module as been designed to minimize its impact on the system and to be as undetectable as possible. However, it has not been conceived to analyze programs with kernel access.

1 Introduction

Today's malware are more and more sophisticated and dangerous. Hence binary program analysis or reverse engineering is an essential but very complex task in order to understand the program, to determine whether it can be harmful or not, and to develop countermeasures. However, many malware, in addition to be malevolent, now implements protections that prevents detection and analysis. Among these protections, there are those that protects against static analysis, and anti-debugging techniques that protects against dynamic analysis. Indeed, many malware include anti-debugging protection that are able to detect the presence of debuggers and change their behavior accordingly. Their analysis then becomes a much more complex task even for the specialist and requires specialized tools that are almost undetectable by standard anti-debugging protections.

In this paper, we present Kolumbo, a dynamic analysis too, that is stealthy and provides many interesting characteristics for facilitating the reverse engineering task.

In Sect. 2, we recall the protections mechanisms that can be used to prevent static as well as dynamic binary analyses. In Sect. 3, we present the existing analysis and debugging tools and compare them to Kolumbo. In Sect. 4, we describe how Kolumbo works.

2 Existing protection techniques

Program or malware's analysis can be divided into two categories:static and dynamic analysis. Static analysis, as its name suggests it, studies the program's behavior by analyzing the binary code. This analysis is performed by using specialized binary editor and similar tools. On the contrary, dynamic analysis studies the program's behavior during its execution. This analysis is done with tools such as debuggers and *ptrace*.

There exists many techniques that a program can use to protect itself against static or dynamic analysis. In this section, we present a few of the more popular techniques that can be used as a protection against analysis.

2.1 Protection against static analysis

Static analysis is done by reading and studying the binary code in order to understand its action during execution.

J. Desfossez (✉)
Revolution Linux, Sherbrooke, QC, Canada
e-mail: jd@revolutionlinux.com

J. Dieppedale · G. Girard
Département d'informatique, Faculté des Sciences,
Université de Sherbrooke, Sherbrooke, QC, Canada
e-mail: justine.dieppedale@usherbrooke.ca

G. Girard
e-mail: gabriel.girard@usherbrooke.ca

Two popular techniques against this type of analysis are code obfuscation and code encryption. Code obfuscation is not studied here since it is not a very efficient nor popular protection against malware static analysis, at least for most of classical malware.

Binary code encryption Binary code encryption using packers is a very popular technique. It works as follows:

- the binary code is encrypted;
- when execution starts, a decoder decrypts the program and stores it in memory;
- execution is redirected to the program which gas just been decoded;
- the decoder is erased from memory.

Variants of this technique exist, such as just in time decoding. However, whatever may be the variant considered, the binary code is never stored under an unencrypted form in a file. Protection against static analysis is then guaranteed. Moreover, this technique can protect any virus against any analysis based on checksum using a pre-established list.

2.2 Protections against dynamic analysis

Dynamic analysis is done by studying the program behavior during its execution. This can be done by using a traditional debugger or a trace tool. These tools perform their monitoring by either inserting breakpoints in the program or by tracing the execution using a special system call. In this section, we present how a program can detect the presence of these tools and then avoid dynamic analysis.

Breakpoints detection In most debuggers, breakpoints are inserted by replacing the instruction at the chosen address by the interrupt instruction *int3* (0xCC). Whenever this instruction is executed, the program's execution is interrupted and the control is given to the parent process, the debugger. The debugger can then analyze the code in memory and once it is done, it replaces the original instruction, re-initializes the program counter and gives the control back to the interrupted program, which can then pursue its normal execution. Unfortunately for these debuggers, breakpoints can be detected by the monitored program. There is many ways to detect the presence of breakpoint. In this section, we present two of these techniques:integrity function and false breakpoint.

Integrity function

A breakpoint can be detected by executing an integrity function that search for the *int3* instruction (0xCC) or do a checksum of the page containing the binary code. The following program shows how a program can detect the *int3* instruction in the *foo* function.

```
void foo()
{
    printf("Hello\n");
}

int main()
{
    if ((*(volatile unsigned *)((unsigned)foo + 3) & 0xff) == 0xcc) {
        printf("BREAKPOINT\n");
        exit(1);
    }
    foo();
    return 0;
}
```

This function is very efficient since a normal debugger relies on a statically inserted first breakpoint (before execution begins) to keep control on the program execution. Kolumbo does not rely on a pre-execution first breakpoint to monitor the execution of a program. The breakpoints can be inserted and removed almost anytime during execution. Hence a first breakpoint can be inserted after the integrity test is done. However this requires an a priori knowledge of the program and a periodic execution of the integrity function can still detect the modification. The breakpoint functionality of Kolumbo must then be used with caution. We will see in Sect. 4 how the internal breakpoint system of Kolumbo works.

False breakpoints

The *int3* instruction is an instruction that can also be used to throw the SIGTRAP signal. So, another way to detect a breakpoint is to use the *int3* instruction and provide a SIGTRAP signal handler to capture the signal and pursue a normal execution. Whenever a debugger monitors the execution of a program that uses the *int3* instruction, the debugger will interpret this instruction as a breakpoint and interrupt the normal execution of the program. The program can detect the fact that its handler was not executed and thus can infer that it is indeed monitored. The following piece of code shows how a program can use a signal handler to detect the presence of a debugger.

```
static volatile int traced = 1;

void sig_handler(int signo) {
    traced = 0;
}

void test_trap() {
    __asm__ __volatile__ ("int3\n\t");
}

int main() {
    signal(SIGTRAP, sig_handler);
    test_trap();
    if (traced) {
        printf("Debugger␣detected␣-␣Signal\n");
        return 1;
    }
    return 0;
}
```

Since Kolumbo does not use the *int3* instruction for breakpoints, it avoids this kind of detection. Instead, Kolumbo uses a system call, the 0x80 interrupt, without any real parameters to force the program to enter into the kernel space. However, even if this approach avoids the signal detection technique, the additional transfers to kernel space may cause other problems. The details of the breakpoint mechanism in Kolumbo are presented in Sect. 4.

***Ptrace* detection** Some traditional *user-space* debuggers, instead of using breakpoints, use the ptrace system call to attach themselves to a process. This specific approach cannot be detected by any of the detection techniques presented earlier. However, since a particular execution can be "traced" by a single process, the monitored process can detect such a debugger by trying to trace itself (`PTRACE_TRACEME`). If the related system call fails, it can supposed that it is being monitored by a debugger.

The following piece of code shows how to detect a trace based debugger.

```
int main() {
    // Ptrace check
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0) {
        printf("Debugger detected - Ptrace\n");
        return 1;
    }
}
```

One of the functionality provided by Kolumbo is to avoid ptrace detection for the debuggers based on this approach. Since Kolumbo works in kernel space, it can modify the behavior of system calls. Hence, whenever a ptrace system call invoked by the monitored process is detected, Kolumbo can modify the return code to avoid detection. With this option, one can use standard tools, such *gdb* and *strace*, to study the behavior of a process. However, a monitored process can bypass this facility by trying to trace itself multiple times. Indeed, a call to *ptrace()* returns normally 0 if the process is already monitored. If multiple calls always return another value than 0, this means that the returned value has been tampered with.

Timing check Another technique that a process can use to detect debugger monitoring is to measure the time taken to execute some pieces of code. A too big gap between the actual execution time and the planned execution time may means that the process is being monitored. These timing tests can be done using the *rdtsc* instruction which returns the value of a 64-bit counter incremented at every clock tick. This counter can provide a very precise measure of the time passed in some code fragment. However, a careful calibration of the threshold is necessary to avoid false positives. The calibration technique and its detection have been presented in this study about detecting HVM rootkits [1].

3 Existing tools

Kernel level program analysis is not a new field. Many tools exist that are supposed to perform exactly this analysis task. In this section, we present few of these tools and compare them to Kolumbo. Our tour is far from being exhaustive since we present mostly the tools for which there exist some public documentation. Indeed many of these tools intend to remain obscure and nearly no documentation is available.

3.1 Helikaon Debugger

The Helikaon Debugger is a Linux kernel module created by Jason Raber [9] from the Riverside Research Institute (RRI). This tool is dedicated to the monitoring of system calls done by a process and the extraction of register contents at some execution location. Its objectives is to be undetectable by the monitored program. The extraction of register contents is done by dynamically injecting jumps from kernel land into an extraction function. Unfortunately, quite no documentation is available for this project outside RRI. However, this platform is very similar to Kolumbo and has inspired many of its concepts.

3.2 Eresi

Eresi [3] is a multi-architecture platform dedicate to binary analysis. Static as well as dynamic analysis is possible through a scripting language. Two components of Eresi, *Ke2dbg* and *Kernsh*, are designed to interact with the kernel. With these components, it is possible to study the executing kernel, to modify it, to debug it, to inject compiled C code and to reroute some kernel functions. Hence it is possible, using the Eresi language, to study any process from kernel space. However it requires a good knowledge of the scripting language.

Even if Eresi can be used to analyze malware, its design was mostly oriented toward analyzing kernel code. In contrast Kolumbo was mostly designed to analyze malware.

3.3 Rr0d

Rr0d [2] is a multi-platform ring 0 debugger that can be used to analyze any binary. Under Linux, a single module is required to install the debugger and a graphic interface is available to interact with the system.

3.4 SystemTap

The *SystemTap* software [5] has been designed to simplify the gathering of information about the running Linux system. It principally targets software developers and its goal is to help in the diagnosis of performance or functional problem.

It provides a command line interface and a scripting language. For the moment, it is mostly useful to monitor applications running inside the Linux kernel. As an example, it can be used to study how drivers behave. The SystemTap ability to monitor user-space applications is quite limited but can be enhanced when it is combined with the utrace tool [8].

3.5 Uberlogger

The Uberlogger [7] software is a Linux kernel module used to monitor some system calls on a particular machine. To monitor system calls, Uberlogger overloads the chosen call directly in the system calls table and stores a trace of all the calls in a database. A web interface is then used to search for the stored system calls and then to analyze them.

This project was principally intended for `honeypots` and `forensic` analysis. Unfortunately, there seems to have no new development on the project in the last three years.

3.6 Linux Trace Toolkit

Linux Trace Toolkit (LTT-ng) [4] is a Linux kernel patch designed to provide precise measures inside the kernel. It is principally used to diagnose performance problems on parallel or real time systems. It comes with a visualization interface (LTTV) that allows tracing of all events on the machine. Since this software is intended to be used on production systems, its developer invested a lot of effort on the software performance and stability.

This is a very interesting tool for Linux server analysis, but its installation is very complex and the information it provides about malware is limited to a list of system calls. This last information is interesting but insufficient for the reverse engineering of malware contrary to Kolumbo's philosophy.

4 Kolumbo operation

In this section, we present the basics of Kolumbo operations and how we perform the system calls redirection. A system call allows a program executing in the user space to call some kernel functions in order to realize operations in need of a higher level of privileges. Among those privileged operations, we can mention different type of accesses to memory, files, devices, etc.

There exists two mechanisms to realize systems calls on the x86 architecture: the first is based on the software interrupt instruction *int 0x80* and the second uses the *sysenter*/*sysexit* mechanism. The *sysenter* approach, integrated since the Pentium II, is a lot more efficient than the interrupt *0x80*. In this section we present in detail how to use the *int 0x80* instruction since it is the only one supported by Kolumbo.

4.1 System calls management

A system call is initiated by entering a 32-bit value corresponding to a kernel function into the *eax* register, the function parameters in the following registers and then executing the *int 0x80* instruction. The interrupt *0x80* transfers the control, in kernel mode, to an interrupt handler in order to treat the system call. The following example shows how to use the system calls *write* and *exit*

```
section .text        ;section declaration
    global _start    ; entry point

_start:
    ;write our string to stdout
    mov     edx,len  ;third argument: message length
    mov     ecx,msg  ;second argument: pointer to message to write
    mov     ebx,1    ;first argument: file handle (stdout)
    mov     eax,4    ;system call number (sys_write)
    int     0x80     ;call kernel

    ;and exit
    mov     ebx,0    ;first syscall argument: exit code
    mov     eax,1    ;system call number (sys_exit)
    int     0x80     ;call kernel

section .data        ;section declaration
    msg     db       "Hello, world!",0xa ;our string
    len     equ      $ - msg ;length of our string
```

4.2 Interrupt 0x80 redirection

Interrupts and exceptions, under Linux, are events that are used to modify instructions execution order. Those events are electrical signals generated by material components inside or outside the processor. We distinguish two types of interrupts: hardware interrupts, generated by input/output devices, and software interrupts, generated either by programming errors (division by zero, segmentation fault, etc.) or by exceptional situations such as page fault or system calls.

For Kolumbo to work correctly, it is necessary to be able to intercept and verify all the system calls executed by a program during its analysis. To do so, we have to replace the interrupt 0x80 handler by our own. Once our handler is in place, we can intercept and analyze any system calls before the system can execute it. In this section, we present how interrupts are managed under Linux and how to replace the interrupt handler.

Handling of interrupts under Linux Whenever an interrupt is triggered, the kernel consults a table, called the *Interrupt Descriptor Table* (IDT), to find the function responsible for it. Each entry of the IDT table corresponds to an interrupt or exception vector which contains an 8-byte descriptor. The CPU register *idtr* is used to locate the IDT table anywhere in memory. It contains the physical address of the table and its maximum size (limit). It is initialized at startup with the assembly instruction *lidt* (*Load IDT Register*).

Localization of the interrupt 0x80 handler To replace the interrupt 0x80 handler, it is necessary to find its address in memory. Thanks to the *idtr* register, we can find the IDT table. It is however necessary to manually compute offsets to find the interrupt 0x80 handler.

The *sidt* assembly instruction (*Store IDT Register*) returns the contents of the *idtr* register in a 6-byte variable: 4 bytes for the IDT table base address and 2 bytes for its maximal size (*limit-field*). Since each descriptor in the IDT table is 8 bytes long, it is necessary to shift the base address to find the descriptor of the actual interrupt. In the case of an interrupt 0x80, we must do a shift of 1024 bytes (8 times 0x80). At this point, we can recover a structure of *idt_descriptor* type that allows us to find the function address.

The following code fragment performs exactly those operations:

```
void *get_system_call(void)
{
        unsigned char idtr[6];
        unsigned long base;
        struct idt_descriptor desc;

        asm ("sidt %0":"=m" (idtr));
        base = *((unsigned long *) &idtr[2]);
        memcpy(&desc, (void *) (base + (0x80*8)), sizeof(desc));
        return((void *) ((desc.off_high << 16) + desc.off_low));
}
```

Replacement of the interrupt 0x80 handler Now that we have found the int 0x80 handler address, let us see how to integrate our code into that one. Our goal here is to replace (in the *system_call* function) the call to the *syscall_trace_entry* and *sys_call_table(,%eax,4)* functions by a call toward our own function. The following code, from the Linux kernel sources, represent the source code of the interrupt 0x80 handler (*system_call*). We can easily identify the call to the two functions we want to replace.

```
ENTRY(system_call)
        pushl %eax                      # save orig_eax
        SAVE_ALL
        GET_THREAD_INFO(%ebp)
                                        # system call tracing in operation
        testb $(_TIF_SYSCALL_TRACE|_TIF_SYSCALL_AUDIT), TI_flags(%ebp)
        jnz syscall_trace_entry
        cmpl $(nr_syscalls), %eax
        jae syscall_badsys
syscall_call:
        call *sys_call_table(,%eax,4)
        movl %eax,EAX(%esp)             # store the return value
syscall_exit:
        cli
        movl TI_flags(%ebp), %ecx
        testw $_TIF_ALLWORK_MASK, %cx   # current->work
        jne syscall_exit_work
restore_all:
        RESTORE_ALL
```

If we take a look at the compiled code, we obtain the following output:

```
c0103d3c:   50                      push   %eax
c0103d3d:   fc                      cld
c0103d3e:   06                      push   %es
c0103d3f:   1e                      push   %ds
c0103d40:   50                      push   %eax
c0103d41:   55                      push   %ebp
c0103d42:   57                      push   %edi
c0103d43:   56                      push   %esi
c0103d44:   52                      push   %edx
c0103d45:   51                      push   %ecx
c0103d46:   53                      push   %ebx
c0103d47:   ba 7b 00 00 00          mov    $0x7b,%edx
c0103d4c:   8e da                   movl   %edx,%ds
c0103d4e:   8e c2                   movl   %edx,%es
c0103d50:   bd 00 e0 ff ff          mov    $0xffffe000,%ebp
c0103d55:   21 e5                   and    %esp,%ebp
c0103d57:   66 f7 45 08 c1 01       testw  $0x1c1,0x8(%ebp)
c0103d5d:   0f 85 c1 00 00 00       jne    c0103e24
c0103d63:   3d 37 01 00 00          cmp    $0x137,%eax   <-- Here
c0103d68:   0f 83 2a 01 00 00       jae    c0103e98      <-- Here

c0103d6e :  # syscall_call
c0103d6e:   ff 14 85 c0 f4 2c c0    call   *0xc02cf4c0(,%eax,4)
c0103d75:   89 44 24 18             mov    %eax,0x18(%esp)

c0103d79 :  # syscall_exit
c0103d79:   fa                      cli
c0103d7a:   8b 4d 08                mov    0x8(%ebp),%ecx
c0103d7d:   66 f7 c1 ff fe          test   $0xfeff,%cx
c0103d82:   0f 85 cc 00 00 00       jne    c0103e54

c0103d88:   # restore_all
c0103d88:   8b 44 24 30             mov    0x30(%esp),%eax
```

From this information and the base address, it is possible to compute the exact location of the calls to the two functions that we are interested in. Once the locations are found, we have to replace the comparison before the jump (*jae*) by a jump to our function. To do this, we must go through the code to search for the *0x0f 0x83* instruction (since it is a unique instruction in this part of the code) and then move 5 bytes backwards. Once at the right place, we replace the *cmp* instruction by a *push* (opcode *0x68*) and then we insert our function address and the *ret* (opcode *0xc3*) instruction. These modifications will provide a call to our function with a minimal impact on the system (only a *push*). Before performing them, we save the addresses and the modified code fragments in order to be able to restore them later.

The following code fragment performs exactly the modifications described above:

```
void set_idt_handler(void *system_call, void *hooked_idt)
{
        unsigned char *p;
        unsigned long *p2;
        p = (unsigned char *) system_call;

        while (!((*p == 0x0f) && (*(p+1) == 0x83)))
                p++;
        p -= 5;
        *p++ = 0x68;
        p2 = (unsigned long *) p;
        *p2++ = (unsigned long) (hooked_idt);

        p = (unsigned char *) p2;
        *p = 0xc3;
```

The same *opcodes* search pattern is applied to find and replace the call to the *syscall_trace_entry* function. At this point, whenever a system call is executed, the function passed

in parameter (*new_idt*) is executed. Our function must first verify that the system call number (stored in *eax*) is lower than *NR_syscalls* (0x137). This is necessary since we have erased this verification due to a lack of space. If this test is satisfied, our function in charge of treating these system calls (*hook*) is invoked. Otherwise, we call the *syscall_exit* function (whose address was found with a procedure similar to the previous one).

The *new_idt* function is used to verify that a system call number is valid through a comparison with the global variable *NR_syscalls*. It is important to notice that we have chosen to program some critical parts directly in assembly language because any changes to a register may provoke a *kernel panic* and that a simple *if* in C affects the registers.

```
void new_idt(void)
{
        __asm__ __volatile__
        (
        "cmp %0,%%eax\n"
                "jae badsyscall\n"
                "jmp hook\n"

                "badsyscall:\n"
                    "jmp dire_exit\n"

        :: "i" (NR_syscalls) // 0x137
        );
}
```

At the end, our *hook* function ensures the link between the system call and the original functions. Our function can then intercept and analyze any system call before it happens. For the moment, the main purpose of the *hook* function is to display the current system call (and its parameters) in case it was triggered by a process under analysis. This function is also used to manage our breakpoint system. This functionality will be discussed later.

Restoring the original interrupt 0x80 handler When our module is unloaded, it is imperative that the system returns to its original state. So, we must restore the interrupt handler in exactly the same state as it was before our modifications. As we have already mentioned, we have saved the addresses and the modified code fragments before any modification. Hence, all we have to do while unloading (function *exit_lkm*) is to restore the saved information.

4.3 System calls redirection

In order to implement additional functionalities, such as monitoring child processes, it is necessary to intercept some system calls. As we have seen in the previous chapter, whenever a call is triggered, the system explores the system call table (*syscall_table*) with the call number (stored in *eax*) to find the right function to call. To intercept a system call, we have to modify this table at the *eax* index to insert the address of the new function to call, to prepare the function to receive the good parameters, to do the treatment and finally to return the right data type.

The main problem is to find the address of the system call table (*syscall_table*). Indeed since the kernel 2.6, the corresponding symbol is not exported anymore. One way to find this address is to read through the *System.map* file generated whenever the kernel is compiled, but this requires a manual action. Moreover, this file may not be present since Linux can execute without it. To resolve this problem, we use the same method as the one used by the rootkit *enyelkm* [6] which dynamically find the address of this table using the IDT table.

Dynamic tracking of the system calls table As seen in the section on the replacement of the interrupt 0x80 handler, it is possible to use the IDT table address to find the code in charge of the system call management. By using a similar technique, we can find the system call table address since it is used in the same code:

```
syscall_call:
        call *sys_call_table(,%eax,4)


c0103d6e:# syscall_call
c0103d6e:ff 14 85 c0 f4 2c c0    call   *0xc02cf4c0(,%eax,4)
```

In this code fragment, we see that the address that we are looking for is 0xc02cf4c0. To find this address, we only have to search for the instruction sequences 0xff, 0x14 and 0x85. The code performing this task is located in the *get_sys_call_table* function.

```
void *get_sys_call_table(void *system_call)
{
        unsigned char *p;
        unsigned long s_c_t;
        p = (unsigned char *) system_call;
        // We look for the sequence 0xff 0x14 0x85
        while (!((*p == 0xff) && (*(p+1) == 0x14) &&
                                (*(p+2) == 0x85)))
                p++;

        dire_call = (unsigned long) p;

        p += 3;
        s_c_t = *((unsigned long *) p);

        p += 4;
        after_call = (unsigned long) p;

        /* 0xfa:cli */
        while (*p != 0xfa)
                p++;
        dire_exit = (unsigned long) p;
        return((void *) s_c_t);
}
```

Some system calls overloading Once the system calls table is found, all we need to intercept (or overload) a system call is to register a function at the address *sys_call_table[eax]*. We have overloaded some system calls in order to add some required functionalities to our module.

As an example, let us study the case of the *fork* system call. In order to recover all the child process identifiers created by the analyzed process, we have captured the *fork* system call (with *clone* and *vfork*). Whenever a *fork* system call is triggered, the original function is called with the same parameters, the return code is saved, a local treatment is done and the original return value is returned. It should be noted here the use of *asmlinkage*, since the system call parameters are passed on the *stack*.

```
asmlinkage int (* zold_sys_fork) (struct pt_regs regs);

asmlinkage int znew_sys_fork (struct pt_regs regs) {
        int ret;
        ret = zold_sys_fork(regs);
        if(search_pid(current->pid) != -1)
                insert_pid(ret);
        return ret;
}

static void syscall_hooking(void) {
        zold_sys_fork = (void *) (sys_call_table[__NR_fork]);
        sys_call_table[__NR_fork] = (void *)znew_sys_fork;
}


static void syscall_unhooking(void) {
        sys_call_table[__NR_fork] = (void *)zold_sys_fork;
}
```

From this moment, the *fork* system call triggers a call to the *znew_sys_fork* function which calls the original function after it has done some local treatment. When unloading the module, it is necessary to call the *syscall_unhooking* function to restore the system call table back to its original state.

Now that we have seen how Kolumbo manipulates system calls and interrupts, let us see how it accesses the memory of a program.

4.4 Virtual memory in Linux

During its execution, a process works only with virtual addresses. This has several advantages such as a better isolation between different processes and the possibility to allocate a contiguous address space even if the system has not enough physical memory. Inside a program, there is no problem to work with virtual addresses. However the processor has no real notion of virtual address space: it just works with physical addresses. The mapping between virtual memory and physical memory is performed in the kernel by the page tables.

Linux uses a three level page table system. The first level, called *Page Global Directory* (PGD), is represented by the data type *pgd_t* and provides access to the second level. This second level, named *Page Middle Directory* (PMD), is represented by the data type *pmd_t* and provides access to the last level. Finally, the last level providing access to the physical page is named *Page Table Entry* (PTE) and is represented by the data type *pte_t*.

It is important to note that every process has its own page table and then the field `mm_struct->pgd` of a task (*task_struct*) can be used to find the PGD of a process. Before accessing a page, it is also necessary to verify that it is available in central memory, otherwise a page fault must be triggered to correct the situation. In the Linux kernel, the *get_user_pages* function is used to access a page in physical memory. This function receives the process identifier, a start address and a size as parameters. It then translates the virtual addresses into physical addresses using the three level page table, ensures that the pages are available, creates an array of pointers toward these pages and returns the number of pages recovered. At this point, it is possible to work on these pages directly from the kernel.

4.5 Process memory segmentation

Whenever a process is loaded into memory, its linear address space is divided into different areas called segments. Typically, there are six main segments: Text, Data, gvar, BSS, Heap and Stack.

These six segments are loaded in three memory areas: *text*, *data* et *stack*. The *text* segment contains the program's binary code:the executable instructions. The *data* segment essentially contains the program's initialized and global variables.

In the kernel, the *mm_struct* structure – which is reachable from the *mm* attribute of a task structure (*task_struct*) – gives us access to these segments. We can also get a process memory segment list in the file *maps* in the pseudo-filesystem *procfs*. For example, when looking at the *maps* file of the *cat* process, we can see:

```
08048000-0804f000 r-xp 00000000 08:01 103586     /bin/cat
0804f000-08050000 rw-p 00006000 08:01 103586     /bin/cat
08050000-08071000 rw-p 08050000 00:00 0          [heap]
b7d60000-b7d9f000 r--p 00000000 08:01 33855
  /usr/lib/locale/en_CA.utf8/LC_CTYPE
...
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
```

Each line of this output provides the following information:

- the *text* segment: easy to find because it has read/execute permissions and is linked to the binary file;
- the *data* segment: read/write permissions and linked to the binary file;
- the *heap* segment;
- one or more dynamic libraries;
- the *stack* segment.

This file is really useful to get an idea of what is going on in memory when the process is running, but it is difficult to only rely on it to follow the evolution of the process during its execution. The following function gives us the start and the

end address of every *Virtual Memory Area* (VMA) loaded by a running process and the address of the three main areas:the *text* area, the *data* area and the *stack*.

```
static void print_mem(struct task_struct *task) {
    struct mm_struct *mm;
    struct vm_area_struct *vma;
    int count = 0;
    mm = task->mm;
    printk("\nThis process has %d vmas.\n", mm->map_count);

    //iteration on process virtual memory areas
    for (vma = mm->mmap ; vma ; vma = vma->vm_next) {
        printk ("Vma number %d: ", ++count);
        printk("  Starts at 0x%08lx, Ends at 0x%08lx\n",
                vma->vm_start, vma->vm_end);
    }

    printk("Code  Segment start = 0x%lx, end = 0x%lx \n"
            "Data  Segment start = 0x%lx, end = 0x%lx\n"
            "Stack Segment start = 0x%lx\n\n",
            mm->start_code, mm->end_code,
            mm->start_data, mm->end_data,
            mm->start_stack);
}
```

Since this function can be called at any time during the execution of the target program, we can decide exactly when we want to print the table. This is very useful when the code of a program can be dynamically modified, rewritten or expanded. For example, when a program is encrypted/compressed by the UPX packer, the unpacking routine creates two new readable/writable/executable memory areas (using the *mmap* system call), decrypts the *text* and *data* segments into these areas, changes the permissions (using the *mprotect* system call) to remove the write permission on the *text* segment and then erases the temporary zone (using the *munmap* system call) used to decrypt the data. All this information was provided by our *print_mem* function and, thanks to this, we know exactly from which memory areas we can get the information we are looking for.

In the following example, we run a packed executable (*hellopacked*) and analyze the system calls with *strace*. For the example, we have only kept the important calls and parameters:

```
$ strace ./hellopacked
...
old_mmap(0x8048000, 50221, PROT_READ|PROT_WRITE|
 PROT_EXEC,) = 0x8048000  # create text area
mprotect(0x8048000, 50218, PROT_READ|PROT_EXEC) = 0 #
 change permissions on the text area
old_mmap(0x8055000, 1103, PROT_READ|PROT_WRITE, ...)
 = 0x8055000 # create data area
mprotect(0x8055000, 1100, PROT_READ|PROT_WRITE) = 0 #
 change permissions on the data area
munmap(0xc01000, 8192)                  = 0 #
 erase the unpacking area
...
```

Reading and writing inside a process memory Now that we know how Linux manages a process memory, let us see how we can access it from the kernel.

From the task structure (*task_struct*), we can access every virtual memory area by following a linked list (via the member *vm_next* of the *vm_area_struct* structure). So, to locate a page, all we need is the task structure of the process we are currently analyzing, the starting address of the target area and its size. The *zget_page* function is designed to find the area that contains the beginning address we passed in argument. Once it is found, it calls the kernel function *get_user_pages*.

```
    int zget_page(struct task_struct *task, unsigned int start,
    unsigned int end, int npages, struct page **pages,
                                    int *page_start) {
        struct vm_area_struct *vma;
        // iterate all vma of the process
        for(vma = task->mm->mmap; vma; vma = vma->vm_next) {
            // found the vma we are looking for ?
            if(start < vma->vm_end && end > vma->vm_start) {
                *page_start = vma->vm_start;
                // get the page(s) and return the number of pages
                return get_user_pages(task, vma->vm_mm, start,
                                                    npages,
                        0, 1, pages, NULL);
            }
        }
        return -1;
    }
```

With this function, we can obtain the pages that contain the area we want to access. Since we cannot read or write directly into physical memory, we first need to map the pages we want in kernel memory with the *kmap* function. This function establishes a link between the physical memory and a virtual address in the kernel space by creating an PTE (*Page Table Entry*) in the kernel page table. When we are done with that area, we must unmap this page with *kunmap*.

For example, the *zdump_region* function copies a sample of a page in an array to work on it later. It receives a page pointer, call *kmap*, reads the data and calls *kunmap*.

```
void zdump_region(struct page *pages, int start, int len,
        unsigned char *data, int offset) {
    int i;
    unsigned char *maddr = kmap(pages);
    for(i = 0; i < len; i++) {
        data[offset + i] = maddr[start + i];
    }
    kunmap(pages);
}
```

Once we know how to read or write into memory, it is quite easy to modify a process segment. In fact, it is as simple as writing in an array of bytes. The following *zchange_bytes* function does exactly that. The main difference between this function and the dump function, lies in the fact that the zchange function is intended to modify an array while the dump function is used to take a copy of an array.

```
void zchange_bytes(struct page *pages, int start, int len,
        unsigned char *data) {
    int i;
    unsigned char *maddr = kmap(pages);
```

```
    for(i = 0; i < len; i++) {
        printk("changing:%02x␣by␣%02x\n", maddr[start + i], data[i]);
        maddr[start + i] = data[i];
    }
    kunmap(pages);
}
```

Now that we have seen how to read and write into a process memory, let us see how to use these features to fetch a program executable code and data from its memory image.

Reconstructing an ELF file from its memory image On Linux, the currently used binary format for executable is ELF (*Executable and Linking Format*). One of its main advantage when comparing to the old *a.out* format is the ability to use dynamic shared libraries.

The ELF structure is quite complex and will not be presented in detail in this paper; but we have to recall the basics. The first 80 bytes are the header of the binary file; we find there the necessary information to understand the file organization. With the *readelf* tool we can print out this header.

```
$ readelf -h test
ELF Header:
Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:                             ELF32
Data:                              2's complement, little
                                   endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              EXEC (Executable file)
Machine:                           Intel 80386
Version:                           0x1
Entry point address:               0x8048130
Start of program headers:          52 (bytes into file)
Start of section headers:          498476 (bytes into
                                   file)
Flags:                             0x0
Size of this header:               52 (bytes)
Size of program headers:           32 (bytes)
Number of program headers:         5
Size of section headers:           40 (bytes)
Number of section headers:         35
Section header string table index: 32
```

When studying this header, it is interesting to note the *Entry point address* and the *Start of section headers* which provides us the address of the segment table. After the header, there are multiple sections corresponding to the program segments presented earlier. Finally, at the end of the file, we can find the section headers table which gives us the length of each section.

So, in order to create a basic ELF, we need the following information:

- The base header;
- The data from *text* and *data* sections;
- The length and the offset of each section.

With the *zdump_region* function we can fetch all of these informations. When dealing with an unencrypted ELF, we have only to copy the *text* and *data* sections to obtain a usable binary since the program is entirely loaded in memory (including the section headers). All we have to do is to find the beginning and end addresses of each section by extracting the information from the *mm_struct* structure. We can compute the same information by reading the ELF headers from the image loaded in memory but doing this introduces some overhead before beginning the copy.

As described earlier, whenever an encrypted program is loaded into memory, the bootstrap routine creates memory regions to store the code and the data of the original program. Once decrypted, the program changes the execution pointer to the new memory area. This type of program raises a problem when comes the time to reconstruct an executable, because the *mm_struct* structure is initialized at the startup of the program and the pointers to the *text* and *data* sections refer to the unpacking program instead of the original program. Moreover, the unpacking method uses the ELF section headers to place the data in the right places, but it does not copy these headers. As a consequence, we have no easy way to determine the end of each section. As of now, when we use the dump mode of Kolumbo, we fetch the good *text* and *data* unencrypted, but the program cannot be executed since the pages are filled with zeros. A manual operation is required to eliminate the unnecessary zeros and thus to obtain an executable program.

Inserting breakpoint dynamically One very interesting concept of dynamic analysis is the ability to stop the program at any given time during its execution to retrieve informations such as its stack or its registers. The commonly used technique for inserting breakpoints is easily detectable by anti-debugging methods.

We have seen, with the trace mode of Kolumbo, that we can print out the registers whenever a system call is triggered. However, since our module works in kernel space, we can obtain this information only when the process executes in kernel space during a 0x80 interrupt. So in order to get the registers between two system calls, we are going to inject a fake system call in the target process. This fake system call is used to simulate a pseudo-breakpoint mechanism in order to get control over the running program and obtain information. Once again we will try to make this procedure as stealthiest as possible.

Here are the steps required to inject a 0x80 interrupt (0xCD 0x80) into the running program:

- Localization of the address where we want to insert the breakpoint (target address), typically between two system calls;
- Localization of the system call which will be used to trigger the insertion of the breakpoint (target system call), normally the last system call before the target address in order to limit the risk of detection;

– Whenever the target system call is triggered, we must:
  - Save the two bytes from the target address;
  - Replace these bytes by 0xCD 0x80;
– Whenever a system call has a return address corresponding to the target address plus two (0xCD 0x80), we know that our pseudo-breakpoint has been triggered, the next steps are to:
  - Print out the registers;
  - Restore the original two bytes saved previously;
  - Change the return address to the target address (minus two from the current return address);
  - Return to the userspace (we do not execute the normal system call procedure, because it is not a system call).

Hence, whenever this fake system call is triggered, we print the information we are interested in, replace the program in its original state, change the return address and resume the process execution. That way the impact is minimal and nearly impossible to detect.

It is important to note that since the breakpoint cannot be placed anywhere in the code, we need to overwrite an instruction and not a parameter otherwise the program will crash.

4.6 Sub-processes monitoring: the fork system call

Until now, we have presented how to trace every system call triggered by a process, fetch its memory and insert breakpoints nearly wherever we want. But if the process we are tracing creates a child, we loose that control. To get around this problem, we have implemented a way to monitor a process family.

Process life cycle Whenever a process starts, the *fork* system call is triggered to create an execution context for the new process. This context includes a memory space, a stack, a kernel environment and a process number (PID). Once inside this new context, the "pseudo-system call" *execve* is triggered to load a new program in the memory space. When the program terminates, it calls the *exit* or *exit_group* system call. The program can also stop when a signal is triggered, for example during a segmentation fault situation.

In Kolumbo, we need to know when new processes are created by a supervised process, and when these processes die.

Monitoring a family of processes from kernel space The *fork*, *vfork* and *clone* system calls are responsible of creating new processes. All three of them return the PID of the new process created. We already know how to overwrite a system call, so let us use this technique to obtain the PID of a new process. The procedure is the following :

– Intercept the *fork*, *vfork* and *clone* system calls;
– If the current process (`current->pid`) is under observation:
  - Call the original function (*fork*, *vfork* or *clone*);
  - Get the return code;
  - Add the new PID to our list of processes;
  - Return the original return value;
– If the process is not under observation, simply return the original return code;

The following code is the modified fork system call which integrate this procedure.

```
asmlinkage int znew_sys_fork (struct pt_regs regs) {
    int ret;
    ret = zold_sys_fork(regs);
    if(search_pid(current->pid) != -1)
        insert_pid(ret);
    return ret;
}
```

We have chosen the simplest approach to store the PIDs: a fixed size array. For now, we suppose that all the child processes have the same behavior but different parameters so we do not need to be exhaustive in our trace.

The normal end of a process occurs when it executes an *exit* or an *exit_group* system call, we have used the same technique of system call overloading to cleanup our process table.

4.7 Avoiding anti-debugging protections

As covered earlier, there are many techniques to detect a debugger. One of the most effective is the one based on *ptrace* which prevent the analyst from using debuggers like *strace* or *gdb*. To use this method, the program tries to call *ptrace* on himself. If it fails, it means that another process already traces the current process.

The next function illustrates the use of this technique.

```
// Ptrace check
if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0) {
    printf("Debugger detected - Ptrace\n");
    return -1;
} else {
    return 0;
}
```

When we execute this function using *strace*, we see that the operation fails:

```
# strace ./detectptrace
...
ptrace(PTRACE_TRACEME, 0, 0x1, 0)       = -1 EPERM
(Operation not permitted)
write(1, "Debugger detected - Ptrace\n", 27Debugger
```

```
     detected - Ptrace) = 27
exit_group(-1)                          = ?
Process 30590 detached
```

We can see that the return code is −1, which means that the program cannot trace itself. The program can then conclude that another process is already tracing it.

Anti-anti-debugging Once again, to avoid the detection based on *ptrace*, we will intercept the system call and change the return value. If the process we are currently tracing tries to trace himself, we will return 1, that way it will believe that no debugger is attached.

If we try the same test program as earlier but with Kolumbo configured, we can see that the *ptrace* system call works and the program continues its execution normally.

```
# strace ./detectptrace
...
ptrace(PTRACE_TRACEME, 0, 0x1, 0)       = 1
exit_group(0)                           = ?
Process 30592 detached
```

This technique is not enough to prevent all kind of *ptrace* based detection, because we return 1 every time. This could be detected if a program tries to trace itself voluntarily twice in order to receive an error the second time. A more complicated scenario could involve two child processes trying to trace the parent expecting an error in one of them.

4.8 Communication between a kernel module and user space

As we have already mentioned, our kernel module needs to exchange information with the user space. To be more precise, we need three types of communication: printing kernel messages in trace mode, making a copy a process address space and configuring the loaded kernel module from user space.

To print kernel messages, we simply use the *printk* function provided by Linux. A character device (`/dev/kolumbo`) is used to make a copy of the process address space. To avoid any buffer overflow, the information is transferred through a *ring buffer* structure.

Since our module provides four different functionalities, it needs to be configured once loaded in the kernel. Hence we have implemented a *sysctl* interface (`/proc/sys/debug/kolumbo`). With this interface we can dynamically change the configuration of the module by setting the desired functionality and providing the required parameters.

5 Conclusion

Kolumbo is an autonomous kernel module designed to help in the analysis of Linux malware. It is equipped with several functionalities oriented against anti-debugging techniques. It is still a new project and some features are currently under development. The main objectives are the support of multi-core architectures, 64-bit kernel and the handling of *sysenter/sysexit* (replacing the system calls).

Kolumbo will become a module designed to assist debuggers (such as *gdb* or *strace*) and make them work when analyzing protect binaries. There are already really good debuggers, so there is no need to create another one, but a little help from kernel space will allow them to trace this kind of binaries.

As of now the functionalities detailed in this paper work on a 32 bits single core machine with Linux 2.6.24. There is still work to do, but it has been successfully used to trace the Linux virus *RST-B*, a SSH scanner which infects the system it runs on. The project can be downloaded via Subversion from sourceforge.[1]

References

1. Desnos, I.A., Filiol, E.: Detection of an hvm rootkit (aka blue-pill-like). J. Comput. Virol. http://www.springerlink.com/content/k7717483424n1h41/. Also available at http://www.esiea-recherche.eu/desnos/papers/hyp.pdf. September (2009)
2. Droids Corporation. RR0D. http://rr0d.droids-corp.org/
3. Desnos, A., Roy, S., Vanegue, J.: Eresi : une plate-forme d'analyse binaire au niveau noyau. In: SSTIC08, Rennes, France. http://www.sstic.org/SSTIC08/programme.do. Also available at http://www.eresi-project.org/wiki/EresiArticles. June (2008)
4. Desnoyers, M., Dagenais, M.R.: Filling the gap between kernel instrumentation and a widely usable kernel tracer. In: Linux Foundation Collaboration Summit 2009, San Francisco, USA. http://events.linuxfoundation.org/archive/lfcs09_desnoyers_paper.pdf. April (2009)
5. Eigler, F.Ch:. Systemtap tutorial. http://sourceware.org/systemtap/documentation.html/
6. eNYe Sec eNYeLKM v1.1. http://www.enye-sec.org/en/
7. Gabes, J., Jamtel, É.L., Alberdi, I.: Uberlogger: un observatoire niveau noyau pour la lutte informative défensive. In: SSTIC05, Rennes, France. http://actes.sstic.org/SSTIC05/UbberLogger/. June (2005)
8. McGrath, R.: utrace: a new in-kernel api for debugging and tracing user tasks. http://people.redhat.com/roland/utrace/
9. Raber, J.: Helikaon linux debugger. In: RECON08, Montreal, Canada. http://recon.cx/2008/index.html. June (2008)

[1] http://sourceforge.net/projects/kolumbo/.