A feasibility study on Ring 0 access and ROP inside virtual machines - type II (hosted)

Krishna Nandula, Bezwada Bruhadeshwar

International Institute of Information Technology.
Hyderabad, Gachibowli.

krishna.n@students.iiit.ac.in

bezawada@mail.iiit.ac.in

***Abstract*—**

Most of the today's x86 computer hardware was designed to run a single operating system and few applications running on it. Around 60% of the resources are underutilized and the enterprises started looking at various optimizations for effective resource utilization. Virtualization is one such technique which was started in the early 60's with sharing of hardware resources among mainframe servers and slowly moved towards x86 architectures.

When we are talking about sharing of resources, it is important to understand various security aspects of virtualization namely memory and process virtualization. This paper targets to perform a feasibility study to understand few of those aspects of type II virtual machine. The study includes understanding and testing trap & emulates functionality of process virtualization using call gate programming and also effects on return-oriented programming on VMs. At the end we will provide some experimental results while performing random tests on virtual machines.

***Keywords*—— Return Oriented Programming, Call Gate Programming, Memory Virtualization, Process virtualization, virtual machine, hypervisor.**

I. Introduction

Motivation: One of the major properties of virtualization [1] is isolation [1]. Any applications which are running inside a guest operating system should not affect the host and any other guests on same machine.

The need to study various security aspects on memory virtualization [2, 8] and process virtualization [2, 8] is growing increasingly as the number of vulnerabilities reported by common vulnerabilities and exposure [3] is rapidly growing.

The main technical contributions of this paper are (1) a study of basic page tables and call gates and setting up the terminology (2) Setting up the base, a way to access Ring 0 inside virtual machine using callgates (3) Setting up the user base, using return oriented programming techniques to by DEP [7] and call Ring 0 function (4) Results on various experimental studies related to performance for various virtual machines.

The rest of the paper is organized as follows. In section 2 a brief review on Call tables mainly GDT [4], WinDDK [5], and OSLoader [6]. In section 3 we look into a simple call gate program and how call gates can be used to running instructions which require higher privileges inside virtual machines. In section 4 we try to overflow a simple program and also use return-oriented programming techniques to bypass the DEP [7] inside virtual machines. In section 5, we would like to present few experimental results obtained during the study.

II. Section 2 , Basic Terminology

A virtual machine is a software implementation of a real machine. A type 2 virtualization is treated as hosted VM, where virtual machine monitor or hypervisor runs beside host operating system. The operating system inside virtual machine is treated as Guest operating system. The complete Guest operating system will be running in user mode i.e., at Ring 3 and any instructions which requires higher privileges will be trapped inside a virtual machine monitor and emulated further.

In order to test this emulation it is important to understand about call tables, call gate descriptors, kernel space. Call tables are used in both user space and kernel space to store the address of routines. If we can replace the call table entry we can reroute the program execution to the function of our choice or execute an arbitrary function with kernel privileges. The call table available in the user space is IAT and at kernel space we have IDT, GDT etc., we are focused on Global descriptor table in this paper.

Global Descriptor Table [4] is a data structure which defines the bases access privileges for certain areas of memory.

Call gate is a special GDT [4] descriptor called system descriptor. These call gates helps to execute Ring0 code from Ring 3. (Less privileged code calling higher privileges).

WinDDK [5] is a windows driver development kit used for building kernel device drivers on Windows operating system.

In the next section we will use these mechanisms and try to execute an arbitrary code at Ring 0 and study the behaviour of virtual machine.

III. Section 3(Part I) , Running code at Ring 0

Virtual machine monitor will take care of the instructions which require high privileges by trap and emulate method. In a normal user mode program, if there is any such instruction VMM automatically emulates. But we need to run our own arbitrary function at Ring 0 to get more control on kernel space.

Caution note: As we are detailing with kernel data structure, there are high chances that any mistake will cause the virtual machine to freeze or the operating system showing blue screen errors.

Below are the basic steps involved in creating any kernel mode drivers.

1. Use windows driver development kit.
2. Write your driver.
3. Test your driver with OSLoader.

As the driver completely executes at Ring 0, we can access the Global Descriptor table add, find if there is any empty entry in the table if there is one add our call gate to GDT . The call gate will have the address of our function routine which needs to be executed at Ring 0.

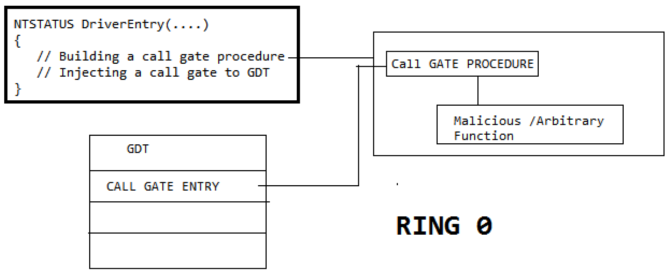
Once the call gate is deployed into the GDT, we need to write a user mode program which will call the function associated to our call gate routine.

Let's briefly discuss the steps involved in build this model.

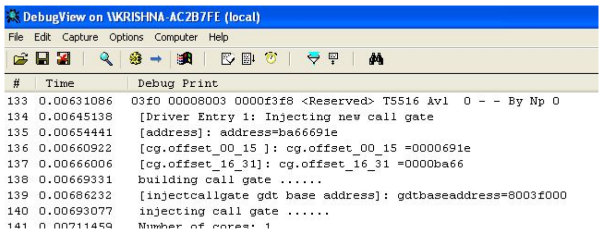
Step 1: Setting up our device driver

Many excellent articles are available which helps us to under how to write our own custom device driver. Windows Driver Development Kit [5] has many samples.

Basic skeleton of our device driver program.



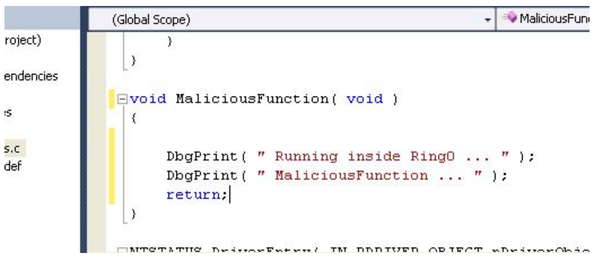
NTSTATUS DriverEntry(....)
{
// Building a call gate procedure
// Injecting a call gate to GDT
}
Call GATE PROCEDURE
Malicious /Arbitrary Function
GDT
CALL GATE ENTRY
RING 0



```
DebugView on \\KRISHNA-AC2B7FE (local)
#    Time         Debug Print
133  0.00631086   03f0 00008003 0000f3f8 <Reserved> T5516 Avl  0 - - By Np 0
134  0.00645138    [Driver Entry 1: Injecting new call gate
135  0.00654441    [address]: address=ba66691e
136  0.00660922    [cg.offset_00_15 ]: cg.offset_00_15 =0000691e
137  0.00666006    [cg.offset_16_31]: cg.offset_16_31 =0000ba66
138  0.00669331    building call gate ......
139  0.00686232    [injectcallgate gdt base address]: gdtbaseaddress=8003f000
140  0.00693077    injecting call gate ......
141  0.00711459    Number of cores: 1
```

Step 2: Our arbitrary function

In the figure below we provide a snap shot of the function which will be executed when the call gate is called.



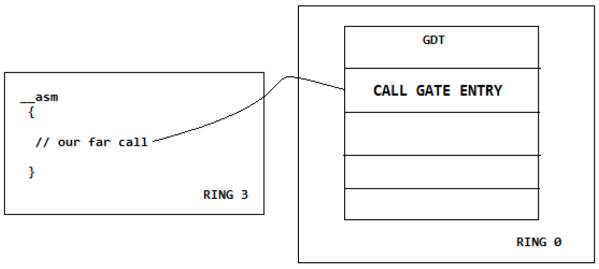
```
        }
}

void MaliciousFunction( void )
{

    DbgPrint( " Running inside Ring0 ... " );
    DbgPrint( " MaliciousFunction ... " );
    return;
}
```

Step 3: Setting up our client program which runs at Ring 3 and invokes our call gate procedure.

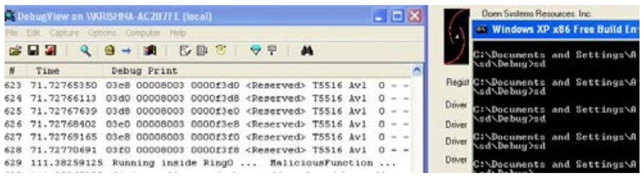
This can be achieved by writing __asm code and using a far call.

Far call is a call to a function which is located in a different segment than the current code segment. Similar to inter segment call.

Basic skeleton of our user mode program and its interaction.



We have successfully gained control on Ring 0, to execute some arbitrary function. The below screen shot show a user programme executed a call gate which in turn runs our Ring 0 arbitrary function.



```
DebugView on \\KRISHNA-AC2B7FE (local)
#    Time         Debug Print
623  71.72765350  03e8 00008003 0000f3d0 <Reserved> T5516 Avl  0 - -
624  71.72766113  03d0 00008003 0000f3d8 <Reserved> T5516 Avl  0 - -
625  71.72767639  03d8 00008003 0000f3e0 <Reserved> T5516 Avl  0 - -
626  71.72768402  03e0 00008003 0000f3e8 <Reserved> T5516 Avl  0 - -
627  71.72769165  03e8 00008003 0000f3f0 <Reserved> T5516 Avl  0 - -
628  71.72770691  03f0 00008003 0000f3f8 <Reserved> T5516 Avl  0 - -
629  111.38259125 Running inside Ring0 ...  MaliciousFunction ...
```

```
Windows XP x86 Free Build En
C:\Documents and Settings\A
\sd\Debug>sd

C:\Documents and Settings\A
\sd\Debug>sd

C:\Documents and Settings\A
\sd\Debug>sd

C:\Documents and Settings\A
\sd\Debug>sd

C:\Documents and Settings\A
```

The function can run malicious code fragments by intercepting system calls and also my acting as a mediator between guest kernel and virtual machine monitor. The user program can be further converted to a shell code and by using the return-oriented programming techniques discussed in next section we can execute the shell code by exploiting vulnerable code.

IV. SECTION 4(PART II) , RETURN-ORIENTED PROGRAMMING

Return oriented programming is a programming technique to bypass the DEP protection provided by windows operating system.

DEP (Data execution and prevention) [7] marks memory pages as nonexecutables, which will not allow the shell code in the stack, heap, or any other memory pools to execute the code.

We use the concept of return oriented programming and try to bypass the DEP protections, by overflowing a vulnerable application and injecting the shell code. The shell code here will be the user mode program written in section III.

Basic steps involved

Bypassing DEP
Converted Binary to Shell Code
Overflow and Shell code execution.

Step 1: Bypass Data Execution and Prevention:

We make a windows function call to VirtualProtect (), which will change the access protection level of a given memory page. This allows us to mark the area of our shell code to make it executable and run the shell code.

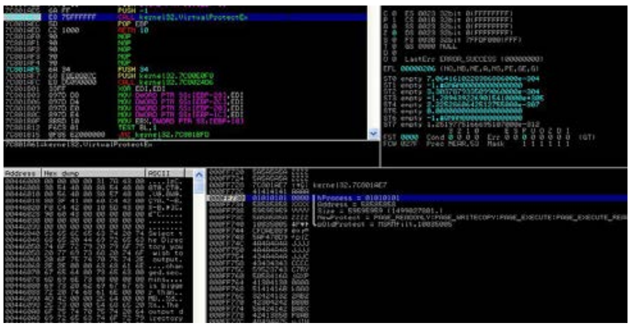
The method VirtualProtect () is present inside kernel32.dll

Sample vulnerable code: (musicplayer.c)

```
int main(int argc, char *argv[])
{
        char buffer[100];
        if(argc>=2)
         strcpy(buffer, argv[1]);
        return 0;
}
```

If DEP is enabled on the machine and the injected shell code after we overflow the above vulnerable program will not be executed.

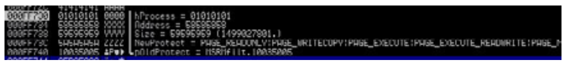
We design our payload and make sure that EIP points exactly to the base address of virtualprotect() method. As a test we debug the code inside immunity debugger and place a break point near virtualprotect () method and overflow the code.



The breakpoint hits in the immunity debugger and manually we try to edit the four parameters required for virtual protect () method to enable the memory area for our shellcode as write executable.

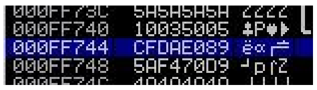
Four parameters of virtualProtect ()

- RET address of virtualproect – This parameter will set the return address of virtualproect() method i.e., the code jumps to our respective address after completing the call.
- Address – This parameter requires the starting address location of the shellcode, as we need our shell code to move from nonexecutable to executable.
- Size – This parameter takes the size of our shell code. “BC 02 00 00”
- New Protect – This is a flag when we set this flag to specify that the memory location needs to be execuable. In our sample we use the value “40 00 00 00”



```
000FF730  01010101  0000  hProcess = 01010101
000FF734  58585858  XXXX  Address = 58585858
000FF738  69696969  iiii  Size = 69696969 (1499027801.)
000FF73C  5A5A5A5A  ZZZZ  NewProtect = PAGE_READONLY|PAGE_WRITECOPY|PAGE_EXECUTE|PAGE_EXECUTE_READWRITE|PAGE_
000FF740  10035005  ♣P♥▶  pOldProtect = MSRMfllt.10035005
```

Once we continue the execution flow, the shell code will be executed by bypassing the DEP protection. This shell code will invoke our call GATE entry in GDT which executes a malicious function in Ring 0. The figure below shows the location of Shell code in the memory.



```
000FF73C  5A5A5A5A  ZZZZ
000FF740  10035005  ♣P♥▶
000FF744  CFDAE089  ëα╒
000FF748  5AF470D9  ┘p⌠Z
```

V. SECTION 5 , COMBINING ROP + RING 0

In section 3, we have discussed briefly on how call gates entry inside GDT will help us to invoke an arbitrary function inside Ring 0. In section 4, using return-oriented technique to bypass data execution prevention mechanisms. We can formulate a complete attack model by combining both of these techniques making it possible to overflow a user level application and executing a shell code which internally makes calls to an arbitrary function in Ring 0.

VI. SECTION 5 , RANDOM EXPERIMENTAL STUDIES

We have performed couple of random experiments to study the behaviour of virtual machines.

Study 1: What if the function inside Ring 0 is calling the same function; user program invokes → Ring 0 functions and the function invokes itself again. Using far call.

Result: Blue Screen Error, the operating systems crashes and restarts itself.

Study 2: Fork Bomb (virtual box)
Configuration plays a keys role while deploy a virtual machine. If a virtual machine is ill configured it can impact the performance of host operating. We studied this with a simple fork bomb [] program under Linux inside Virtual Box.

Command executed : () { :| :& }; :

Result: As the virtual machine is ill configured the fork bomb completely uses the memory allocated and also affects the host operating system performance. Finally crashing the virtual box.

Stud 3: memtest86+ (virtual box)
As a random test on memory we initiated the memory test feature available in backtrack and found that the virtual box crashes during the memory test.

Results: Virtual Box crashed during the memory test.

The above tests randomly crashed the virtual machines. The work is in progress to analyse the reasons behind these crashes.

VII. FUTURE WORK

Using these mechanisms as a base, we further continue our research to study various virtual machine kernel/micro kernel data structures. A deep dive into I/O virtualization to understand possible security risks inside virtual machines.

VIII. CONCLUSIONS

Virtual machines runs both guest os and its application in user mode , we have provided a combination of Running code in Ring 0 using call gate + Return Oriented Programming techniques to bypass DEP protection and invoke a function at Ring 0. We have successfully showed a way that even a virtual machine runs the guest OS in unprivileged mode. We can execute our own function routine to run at Ring 0.

We further continue to study various different areas of virtual machine using Ring 0 function.

REFERENCES

[1] Jiang Wang, Sameer Niphadkar, Angelos Stavrou, and Anup K. Ghosh "Virtualization Architecture for In-depth Kernel Isolation"

[2] Ken Barr, Ravi Soundararajan, Carl Wald Spurger (VMware) "Introduction to Virtual Machines"

[3] Common Vulnerability & Exposure, 535 "http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=vmware"

[4] Global Descriptor Table or GDT data structure "http://en.wikipedia.org/wiki/Global_Descriptor_Table"

[5] Windows Driver Kit by Microsoft for device driver programming http://msdn.microsoft.com/en-us/library/windows/hardware/gg487428.aspx

[6] OSLoader, the software used to test our written kernel programs.

[7] A detailed description on Data Execution & Prevention "http://support.microsoft.com/kb/875352"

[8] Chen, P. M. and Noble, B. D. 2001. When Virtual Is Better Than Real. In Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (May 20 - 22, 2001). HOTOS. IEEE Computer Society, Washington, DC, 133.

[9] R. Hund, T. Holz, and F. Freiling. Return-Oriented Rootkits: Bypasssing Kernel Code Integrity Protection Mechanisms. In Proc. of the 18th USENIX Security Symposium, 2009

[10] Adams, K. and Agesen, O. 2006. A comparison of software and Hardware techniques for x86 virtualization. SIGARCH Comput. Archit. News 34, 5 (Oct. 2006), 2-13.

[11] Sina Bahram, Xuxian Jian, Zhi wang, Mike Grace, Jinku Li,Deepa Srinivasan "DKSM : Subverting virtual machine introspection for Fun and Profit"

[12] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection through VMM-based"Out-of-the-Box" Semantic View Reconstruction. In Proc. of the 14th ACM CCS, 2007.

[13] Travis Ormandy "An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments"

[14] Joanna Rutkowska, "Red Pill" , http://invisiblethings.org

[15] Peter Ferrie "Attacks on Virtual Machine Emulators" , Symantec Advanced Threat Research

[16] Keith Adams, Ole Agesen "A Comparison of software and hardware techniques for x86 Virtualization" vmware.

[17] Dirk Leinenback "Communicating Virtual Machines" . Saarland University. Germany.

[18] Steven Hand,Andrew Warfield,Keir Fraser,Evangelos Kotsovinos, Dan Magenheimer "Are Virtual Machine Monitors Microkernels Done Right ?"

[19] Stephen Checkoway, Hovav Shacham "Escape from return-oriented programming"

[20] Ryan Riley, Xuxian Jian, Dongan Xu "Guest-Transperant Prevention of Kernel Rootkits with VMM-based memory shadowing"

[21] Sherri Sparks, "Shadow Walker – Raising the bar for Rootkit Detection"

[22] Joanna Rutkowska "Introducing Blue Pill" http://theinvisiblethings.blogspot.in/2006/06/introducing-blue-pill.html

[23] Neil MacDonald "Hypervisor attacks in the Real World"

[24] G.Carrette, crashme tool for testing the robustness of the operating system. http://people.delphiforums.com/gjc/crashme.html

[25] Virtual box http://www.virtualbox.org an open source QEMU based virtual machine and vmware http://www.vmware.com .

[26] Tom Liston, presentation on "Thwarting Virtual Machine Detection" http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf