# How to Protecting Kernal Code and Data

Nisar Ahmed, Muhammad TehseenQureshi, Hafiz YasirRana BadarqaShakoor
*Mohi-ud-Din Islamic University, Nerian Sharif, AJ&K, Pakistan*
*Department of Computer Science, University of Agriculture, Faisalabad, Pakistan*
*Department of Computer Science, Global Institute Lahore, Lahore, Pakistan*
*Department of Computer Science, University of Agriculture, Faisalabad, Pakistan*

**Abstract—**Most of the computer having Von Neumann Architecture nowadays facing the risk of attack of modifying kernel code. It is due to same addresses space of kernel code as well as data. In this paper, we discuss the methods to protect kernel code as well as data. The main theme behind this paper is how to make secure kernel code.

***Keywords****—Operating System; Kernel; Code; Data; Kernel Protection.*

## I.INTRODUCTION

Two commonly used ways to make kernel codes ecureare. The operating system provides itself or some system reinforcement established by some scholars. This method take some hard ware benefit (on/off status of bits in segment table and page table etc) to protect every inch of important areas of memory. This ideal though works up to some extent to secure kernel code but there are some boundaries when fronting the risk of kernel rootkits. For example, by the use of 32-bit machine via NX bit, one must set some legacy CPU not maintained with PAE. Even we make bits on using this method, attackers can break this security. So that's why this method is not so good to protect the kernel code.

There is another conventional method called kernel code integrity checking. It based on complicated strategy and complex method of detecting the change in kernel code. It also does nothing to prevent the malicious code of attackers. It may detect some modifications in kernel code but this method does not know how these amendments are prepared and do nothing for analyzing the imposition and its retrieval.

We recommend a different tactic to secure the code of the kernel. Harvard memory architecture implemented to it. It takes a separate physical memory area for kernel code and separate for data. It ensures after booting of operating system, if some illegal operation is performed on code of kernel, it will be conveyed to some memory's shadow which is designed to coverall prohibited tasks. This tactic is applied in Operating system and it can secure the code of kernel unfluctuating the attackers have got the upper most level of privileges. It also archives all prohibited tries to alter code of the kernel during illegal operations started which provides help ful analysis report about intrusion. Attackers do not guess that their illegal attempts have been captured by System. This method takes some advantage of hardware structures and simply enforces very trivial overhead.

There creation of papers helters as follows. Part II covers the risk model of paper. Part III sates design of the system design and execution facts. The assessment of our tactics presented in part IV. Part V describes the limitations and future work. Part VI stretches transitory explanation to associated work and part VII is conclusion.

## II. RISK MODEL

The active OS stuffs laden in memory are modified by late stroot kits. Convention ally root kits do not alter the code of the kernel directly. They add spiteful code in the area of kernel data and then that malicious code executed. In the same way some root kits also have the volume to alter the kernel code. The unsafe kernel of Linux do brk allows the muggers to alter any pages bits of privilege in the system that give access to alter any OS kernel's data structure or code. Some changes allow modifying physical address related with respect to a virtual address. If this modification is allowed to root kits, severe damages can occur in operating system.

There are fewer attackers who are adopting the traditional approach of code injection. More attackers modify kernel code and other modern techniques. Therefore it is very necessary to protect kernel code. Root kits are our threat model in this paper.

## III.DESIGN OF SYSTEM AND ITS IMPLEMENTATION

### *A. An Architectural Tactic*

Harvard architecture take start from Harvard Mark I computer. The Instruction and data occupied separates paces and trails to CPU in this technique.

By using of von Neumann technique in the Utmost current computers where data and code use identical address space. The sharing of same address space by instructions and data is root cause of injection of malicious code. The Harvard architecture provides the advantage of separate address space for both data and instructions. Atupper most level Code cannot be used as data and further moses facts cannot bet reated as instructions of privileges to attackers. The code injection becomes almost impossible in this way.

Some features of Harvard architecture are similar like Von Neumann architecture such as separate data / instructor cache and separate data / instruction TLBs (translation look a side buffer). These feature help us to make kernel more secure.

The simplest way to implement Virtualization is by using VMH. Though prominence to pen source Operating system (Xen [3] and KVM [8] still under progress. They have not widely accepted. Most of their features are not tested. In comparison, Linux is widely accepted and well reputable to highest ability. That's why we adopt to build our virtual environment in operating system rather than in VMM.

### B. *Implementation Details*

In our system, described infig2, Vis the linear address space of kernel code). We have:

For instruction fetch:

Mapping_i(va)=$pa_{exec}$…………….(1) For

Data access:

Mapping_d(va)=$pa_{exec}$…………………(2

$pa_{exec}$€P_1, $pa_{data}$€P_DandP_1andP_D={}
P_I is for instruction physical address space P_D is the Physical address special so called shadow memory.

Once the method of protection adopted, it will ensure the security of kernel code while in operating system working. Normally execution of an instruction is steps process which include fetch, decode, execute, memory and write back. The separate cache and TLBs are introduced to cover the two steps (fetch, memory) which do not interfere each other.

The two corresponding PTE sinI-TLB and D-TLB of aparticular linear address should be similar but there is no way to ensure it. If we insist operating system to provide two different page tables respectively for fetch and memory steps, then the malicious code operation to kernel and normal execution of kernel will be according to contents of different physical pages. In the same methodology, Mapping_I and Mapping_ d are implemented. The steps are given below.

- When the system is started, al locate a continuous area (called shadow memory) whose size is equal to kernel code region. It is pointed by a pointer variable Kernel _text _mirror. Kernel code is copied there. There is a one to one correspondence between the frames of shadow memory and the frames of kernel code. Shadow (ppn)=ppn+(mirror_start_text_start) (3) text_ start is first frame number of the kernel code, mirror_ start is the first frame number of the shadow memory, both memory are related with each other.
- Inter change the ppn address of all kernel code with shadow memory address temporarily, and then load them to D-TLB.
- Recover all the Kernel code bits that have been replaced, and interchange the R/Wbitto0.
- If the page fault occurs by over writing the kernel code, modify the page fault handler of Linux, modify the corresponding bit to1, then go to step 2and3.

The bits of Kernel code pages are in TLB for their global bit is set commonly. If these pages are modified or deleted accidently, they are set to read only, page fault will be generated and double map in go f address translation will occur against step 4 by page fault handler.

This approach made little bit change in source code of page initialization and page fault handler. The changes of new code are not more than 200 hundred rows and checked and analyzed again and again to adjust with kernel code and work properly.

## IV. EVALUATION

### A. *Performance impacts*

We are using our experimental plate form on Pentium IV 3.0 G processor with 512 MB of RAM. In our first experiment, the operating system we use is Fedora Core 4 with a vanilla Linux kernel. The second experiment use same system with modified kernel code. Our studies prove that there are no substantial differences in performance after changing the kernel code in system calls and facilities of operating system. Little bit difference is negligible. The full use of TLB is the cause of negligible performance loss.

In the same way, the impact of kernel code in D_TKB is minimal. The entries of PIV's TLB is as many as 128 the kernel code page's only take few of them and does not show any over head on performance.

### B. *Case Study*

We stretch awareness of new root kit by changing adoring 0.56 to check the efficiency of system in our review. The root kit is a kernel root kit on Linux. It changes the entry address and also uses few system calls. Runs over Linux new changed Ador-ng0.56. It interferes with system calls codes by modifying the contents of their pages to 0 by this OS break down.

We installed the altered code on the Fedora Core 4 with Vanilla Linux deprived of our security technique, the system Become crashed after installation of root kit in our experiments. We installed altered kernel code, the root kits installation quiet succeeded on the same machine. The working of machine and no error mentioned.

We checked the contents of file system. Found the value of kernel_text_mirror and copy the shadow memory contents by map. We can watch the contents of shadow memory we real zeros which show that any attack to change kernel code redirected to shadow memory.

Our mechanism will redirect the attacker's root kits to additional area, thus makes the ineffective and records their tries.

## V. LIMITATIONS AND FUTURE WORK

Garfunkel and Rosenblum proposed VMM over IDS. OS had compromised when it start working. A Revirt which allows intrusion investigation by virtual machine logging and replay VMM based system implement by SamuelT. King. Its working shows that consuming virtualization for security determinations is useful.

RyanRileyadvisedatechniquetostopcodeinjection occurrenceatuserlevelbyvirtuallydividingmemory.Itis alike to our technique but there are some flaws in his technique. Single step mode is mostly used for alter the contents in TLB, due to fast context switch of user processes its lowdown instruction execution and cause of big over head and the performance.

Wursteretal. Pass over self-check-summing appliesadis similar address translation method. NathanE.Roshenblumma dean extension to Xenhy per visor to apply context sensitive paging mapping forth common resolution. After altering the target program: they used common technique, they turned on the technique of situation penetrating mapping to stretch the self-checks umming am is apprehension that the code of the program is none altered.

## VI. CONCLUSION

We can make better the security of Linux kernel by implementing some useful features of other architectures to current X86. To save code of kernel, the virtualization technology to apply a Harvard memory use in this review. Von Neuman architecture used ford signed OS architecture, implements identical some over head and archives the spiteful attacks to change code of kernel. This tacticisuseful, easy and very effect as shown by experiments.

## VII. REFERENCES

[1] Z. Bai, L. Wang, J. Chen, L. Xu, J. Liu and X. Liu, "DTAD: a Dynamic Taint Analysis Detector for Information Security," The Ninth International Confernce on Web-Age Information Management, vol. 2,591-597, 2008.

[2] J. Sun, X. Li, H. Chen and H. Tan, "A Virtualized Harvard Architectural Approach to Protect Kernel Code," First International Workshop on Education Technology and Computer Science, 1020-1024, 2009.

[3] G. Junkai, J. Weiyong, "An Approach for Sensitive Binary File Protection," International Forum on Information Technology and Applications, 716-718, 2009.

[4] H. Mohanty, M. VenkataSwamy, S. Ramaswamy, R.K. Shyamasundar, "Translating Security Policy to Executable Code for Sandboxing Linux Kernel," Third UKSim European Symposium on Computer Modeling and Simulation, 124-129, 2009.

[5] S. Butt, V.Ganapathay, M.M.Swift and C. Chang, "Protecting Commodity Operating System Kernels from Vulnerable Device Drivers," Annual Computer Security Applications Conference, 301-310,2009.

[6] X. Jiang, Y. Solihin, "Architectural Framework for SupportingOperating System Survivability," NSFAward CCF, 457-465, 2011.

[7] J. Li, Z. Wang, T. Bletsch, D. Srinivasan, M. Grace and X. Jiang, "Comprehensive and Ef fi cient Protection of Kernel Control Data," IEEE Transactions on Information Forensics and Security, vol. 6(4),1401-1417, 2011.

[8] B. Blackham, Y. Shi, S, Chattopadhyay, A. Roychoudhury and G.Heiser, "Timing Analysis of a Protected Operating System Kernel,"2011 IEEE 32nd Real-Time Systems Symposium, 339-348, 2011.

[9] D. Stanley, D. Xu, E.H. Spafford, "Improved Kernel Security ThroughMemory Layout Randomization," IEEE, 2013.

[10] F. Yamaguchi, N. Golde, D. Arp and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," 2014 IEEE Symposium on Security and Privacy, 590-604, 2014.