Kernel Integrity Protection from Untrusted Extensions Using Dynamic Binary Instrumentation

Akshay Kumar Peter Goodman Ashvin Goel Angela Demke Brown
University of Toronto
ak.kumar@mail.utoronto.ca, pag@cs.toronto.edu, ashvin@eecg.toronto.edu, demke@cs.toronto.edu

1. Problem Statement

Device drivers are the major source of concern for maintaining security and reliability of an operating system. Many of these device drivers, developed by third parties, get installed in kernel address space as extensions. These extensions are implicitly trusted and are allowed to interact with each other and kernel through well-defined interfaces and by sharing data in an uncontrolled manner. Unfortunately, the assumed trust leaves commodity OSes vulnerable to misbehaving and malicious kernel extensions.

Over the years, researchers have attempted to address the problem of kernel extensions misbehaving and potentially compromising the kernel. This problem has been approached using compile-time and run-time checking/annotations [4, 8, 14, 15], hardware protection domains [7, 14], by moving extensions into user space [6, 15], and by extending virtual machines (VMs) and hypervisors with the ability to introspect and monitor a running kernel and its extensions [12, 13, 16]. While these approaches have already achieved many of the goals of this research, all suffer from at least one of the following issues: i) they depend on VMs/hypervisors, and thus limit the breadth of extensions which can be monitored, or; ii) they require a custom compilation toolchain and/or kernel modifications, and so limit the likelihood of their integration with existing OS build processes.

Clearly, a high level of security against misbehaving kernel extensions can be achieved and is desirable. We think that the best way to encourage the adoption of secured kernel extensions by existing commodity kernels is by showing that arbitrary extensions can be securely executed. Our approach uses dynamic binary instrumentation (DBI) to achieve this goal [2, 3]. We have implemented a proof-of-concept that enforces control-flow integrity (CFI) constraints [1] in the Linux kernel using the DynamoRIO Kernel (DRK) DBI framework [9, 10]. Our choice of DRK as a host platform was motivated by its ability to provide fine-grained control over instrumentation. Our prototype is loaded as a kernel module and transparently monitors all control-flow transfers while the kernel is running. Our system is able to detect and report unauthorized control transfers by potentially malicious kernel modules. Building on this, we intend to develope a security system which will protect the kernel from multiple type of malicious activities by untrusted extensions. We are hopeful of achieving high level of security with a smaller performance overhead by instrumenting only extension binaries and isolating it into different domain monitoring all its interactions with the kernel. However achieving this is difficult for commodity operating system since OS kernel & its extensions reside in same address space and DRK, having complete control over all kernel execution, holds little clue about when these extensions start executing. We proposes to tackle this by dynamically inserting unpassable barriers between the kernel and its extensions using hardware page protection in a similar fashion to *Nooks* [14] and *Gateway* [13]. Any transition from kernel page table to extension page table will generate a page fault leading to start the instrumentation system. Once started the instrumentation system will get "attach" to the untrusted extensions binaries instrumenting it and putting it into code-cache. The code cache will get "filled in" in two ways: a basic block is eagerly inserted into the code cache when that basic block is the target of a direct branch, and lazily inserted to the code cache when that block is the target of an indirect branch. Direct and Indirect branches that transition from the code cache to the kernel will "detach" the instrumentation system by branching it to the desired address if that address is a trusted entry point. Once loaded into the Linux kernel, our system will replace Linux's exception and interrupt handlers and the system call table and any future modifications to these subsystems will be disallowed.

Our system will contain an Interface layer which will mediate communication between kernel and instrumented extensions. It will guarantee control flow integrity (CFI) by enforcing following conditions: i) Maintaining kernel code integrity ii) All *call* and *jump* instruction between extensions and kernel should target to trusted entry point and iii) All *return* instruction should target to the instruction following call. Our system will maintain kernel code integrity by providing page protection against write on transition from kernel code into the code cache. The direct access of kernel code page from code-cache will lead to a page fault and it will get handled by our system. The other approach to maintain kernel code integrity is by sandboxing every extension instructions so that it can't modify kernel code directly. This approach will save us the cost for page fault handling but it will increase instrumentation cost. All control transfer happening due to *call* and *jump* instruction will get mediated by interface layer. It will verify the target address for these control transfers by ensuring if the address belongs to kernel page table or extensions page table and whether this address is a trusted entry point or not. We intend to reduce the performance overhead cause due to mediation by creating a fast path for all direct calls from untrusted extensions to kernel during instrumentation. Untrusted extensions can also subvert CFI by manipulating control data in its stack frame causing return-oriented attacks. Several system achieves stack integrity by using stack canaries[5] or return stack encryption[11]. However these approaches are not very helpful for us since we intend to monitor execution at Interface layer

leaving kernel execution unaffected. To protect the kernel from such attacks our system will monitor the sequence of all *call* and *return* happen accross Interface layer. It will record the return address at the top of the stack on every call before switching to extension page and on subsequent return it will verify if it is targeted to previously stored address or not.

As mentioned previously, the system will maintain kernel code integrity by making it write protected when accessed from code-cache. However there are still security sensitive data in kernel space which can be exploited by malicious modules to launch attacks. Maintaining data integrity in kernel space is difficult as they remain widely scattered across kernel memory space. Existing system uses different approach to maintain data integrity. XFI [8] implements an inline monitoring system through guarded write instructions where as BGI [4] maintains data integrity by maintaining an access control list at byte-granularity level and inserting permission checking code with every memory access operation during compilation. we have different approaches to implement kernel data integrity in our system and one of them is providing page protection to kernel data against write and execute on transition from kernel to code-cache and handling it at Interface layer. This approach can prove costly for us when extension will do lot of data operations. The other approach is by inserting access check instructions with every memory reference operation during instrumentation and verifying it against an access control policy but this will increase instrumentation cost for us. For implementing kernel data integrity it is important to maintain a judicious tradeoff between performance and kind of security system enforces. We are still open to our approach of implementing it in our system.

2. Related Work

The system proposed by us draws its inspiration from volume of previous research work on device driver isolation. One of the method to achieves it in kernel is by pushing them in user address space. *Microdrivers* [15] splits the extensions in kernel and user space component using programmer annotations. This requires an effort from programmers to decide which part of extensions they want to run in user-space. The system also dont achieve full isolation as a part of extensions runs in kernel address space. *Nexus* [6] is micro-kernel based system and executes the extensions in user space. It uses reference validation mechanism to isolate extensions from kernel space. It enforces safety properties beyond what previous user-level driver system provide. These systems are not very effective as they dont support all extensions and poses considerable performance overhead.

There are systems like *Gateway*[13] and *HUKO*[16] utilizes virtualization to establish strong isolation between kernel components but they are not effective when it comes to isolating native device drivers. Other systems implements isolation of malicious extensions while running it in kernel address space. *Nooks* [14] uses hardware enforced protection domain to isolate the untrusted extensions in kernel space. The use of hardware protection layer poses high performance overhead in it. This is because it doesn't contain support for accessing global data structure directly. *Mondrix*[7] uses similar approach to provide fine-grained memory isolation for unsafe kernel extensions with low overhead but it requires a specific designed processor architecture to support the protection domain. *Nooks* and *Mondrix* enforces strong isolation in kernel space but they are not effective in protecting kernel from malicious extensions. *XFI* [8] makes use of software fault isolation (SFI) technique to enable a host program safely execute untrusted extensions in its address space by enforcing control flow and data integrity. *Byte-Granularity Isolation (BGI)* [4] is another approach which enforces fault isolation by implementing memory protection through access control. It associates an access control list with each byte of virtual memory to control the memory access by untrusted extensions. The untrusted extensions in BGI is provided temporary access to the kernel data structure by wrapping up all the calls with a permission granting and permission revoking code.

Both XFI and BGI are powerful technique of fault isolation in kernel space but they lack transparency. XFI uses binary rewriter which uses debugging informations to insert software guard that perform checks at runtime whereas BGI requires a special compiler to compile the source code which inserts permission checks before every memory access and control transfer statement. Our system is inspired by BGI and we are hopeful of achieving same level isolation in kernel space maintaing transparency and system performant.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Transactions on Information and System Security*, 2007.

[2] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph.d. thesis, Massachusetts Institute of Technology, 2004. URL `http://portal.acm.org/citation.cfm?id=1087758`.

[3] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*, pages 265–275, San Francisco, CA, 2003. ACM. URL `http://portal.acm.org/citation.cfm?id=776261.776290`.

[4] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast Byte-Granularity Software Fault Isolation. In *SOSP*, pages 45–58, Big Sky, MT, 2009. ACM.

[5] C. Cowan, D. M. Calton Pu, H. H. J. Walpole, S. B. Peat Bakke, P. W. Aaron Grier, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.

[6] P. R. Dan Williams, E. G. . n. S. Kevin Walsh, and F. B. Schneider. Device driver safety through a reference validation mechanism.

[7] J. Emmett Witchel and R. K. Asanovi. Mondrix: Memory isolation for linux using mondriaan memory protection. Brighton, UK, 2005. SOSP.

[8] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software guards for system address spaces. In *OSDI*, pages 75–88, Seattle, WA, 2006. USENIX Association.

[9] P. Feiner, A. Demke-Brown, and A. Goel. A Design for Comprehensive Kernel Instrumentation. In *HOTDEP*, Vancouver, Canada, 2010. USENIX Association.

[10] P. Feiner, A. Demke-Brown, and A. Goel. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *ASPLOS*. ACM, 2012.

[11] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. Published in USENIX Security Symposium '01, 2001.

[12] D. Hofmann, R. Kim, and Witchel. Ensuring operating system kernel integrity with osck. ASPLOS, 2011.

[13] A. Srivastava and J. Giffin. Efficient monitoring of untrusted kernel-mode execution. *Work*, 2011. URL `http://www.cc.gatech.edu/ giffin/papers/ndss11/SG11.pdf`.

[14] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *TOCS*, 23(1), 2005. ISSN 0734-2071.

[15] M. J. R. Vinod Ganapathy, M. M. S. Arini Balakrishnan, and S. Jha. The design and implementation of microdrivers. Seattle, Washington, USA, 2008. ASPLOS.

[16] D. T. Xi Xiong and P. Liu. Practical protection of kernel integrity for commodity os from untrusted extensions. San Diego, CA, USA, 2011. NDSS.