Analyzing and Improving Linux Kernel Memory Protection: A Model Checking Approach

Siarhei Liakh
North Carolina State University
sliakh@ncsu.edu

Michael Grace
North Carolina State University
mcgrace@ncsu.edu

Xuxian Jiang
North Carolina State University
jiang@cs.ncsu.edu

ABSTRACT

Code injection continues to pose a serious threat to computer systems. Among existing solutions, $W \oplus X$ is a notable approach to prevent the execution of injected code. In this paper, we focus on the Linux kernel memory protection and systematically check for possible $W \oplus X$ violations in the Linux kernel design and implementation. In particular, we have developed a Murphi-based abstract model and used it to discover several serious shortcomings in the current Linux kernel that violate the $W \oplus X$ property. We have confirmed with the Linux community the presence of these problems and accordingly developed five Linux kernel patches. (Four of them are in the process of being integrated into the mainline Linux kernel.) Our evaluation with these patches indicate that they involve only minimal changes to the existing code base and incur negligible performance overhead.

1. INTRODUCTION

Despite years of research, code injection attacks continue to be one of the major ways of computer break-ins and malware propagation [21]. Specifically, a code injection attack is a method whereby an attacker inserts malicious code into a running process and transfers execution to the malicious code (e.g., by hijacking its control flow). After that, the attacker can gain control of a running process and carry out other malicious activities, including the installation of bot programs for remote control and the modification of system files to allow for unauthorized access, etc.

There exist a variety of solutions [15, 25, 28, 30, 33] to deal with code injection attacks. Among the most notable, $W \oplus X$ [1] is a scheme that has been proposed to counter code injection attacks. In essence, $W \oplus X$ enforces the following property, "a given memory page will never be both writable and executable at the same time." The basic premise behind it is that if a page cannot be written to and later executed from, code injection becomes hard, if not impossible, to launch. Due to its effectiveness in defending against code injection attacks, since its proposition, $W \oplus X$ has been widely adopted in commodity OSs (e.g., Windows and Linux). Hardware vendors such as Intel and AMD also follow up this scheme by providing necessary hardware support (in the form of NX support [9]) to facilitate the $W \oplus X$ enforcement.

From the OS kernel perspective, establishing and maintaining the $W \oplus X$ property requires a sound design. In this paper, we look into the Linux kernel and analyze the way it takes to protect its own kernel memory. This is important as the Linux kernel is typically a part of trusted computing base (TCB) in existing solutions to defend against code injection attacks. In our analysis, we took a model checking approach so that we can take advantage of its power to rigorously examine the soundness and completeness of $W \oplus X$ enforcement in Linux kernel. More specifically, we first build a model of Linux kernel memory management subsystem and then apply model checking to verify the $W \oplus X$ property. In case of violation, model checking has the unique advantage in accurately pinpointing potential problems in the current Linux kernel design and implementation.

We have successfully developed a Murphi [10]-based abstract model to analyze linux kernel memory management. Based on our modeling, we were surprised to discover several issues related to Linux kernel memory management: (1) First, the current Linux kernel does not strictly separate the kernel code and kernel data, immediately leading to $W \oplus X$ violation. (2) Second, as part of its implementation, Linux kernel promotes creation of multiple virtual aliases, potentially with conflicting permissions, for the same physical memory page, leading to exploitable scenarios for kernel data execution or kernel code modification.

We have confirmed with the Linux kernel community the presence of these issues. We have also accordingly developed kernel patches to fix these problems and these patches [17, 18, 19, 20] are in the process of being integrated into mainstream Linux kernel.[2] Our evaluation indicates that these patches are compatible with existing Linux kernel code base and contain minimum modifications to the existing interfaces that manage kernel memory. We also observe that

[1]Strictly speaking, the property is $\neg(W \wedge X)$, but we chose to use the traditional $W \oplus X$ notation to emphasize mutual exclusivity of write and execute access.

[2]For the convenience of kernel patch debugging and adoption, we have developed five smaller patches in total (Section 4). Four of them are being tested for final integration into mainstream Linux kernel and the remaining one is still being internally assessed for suitability in mainline Linux kernel.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

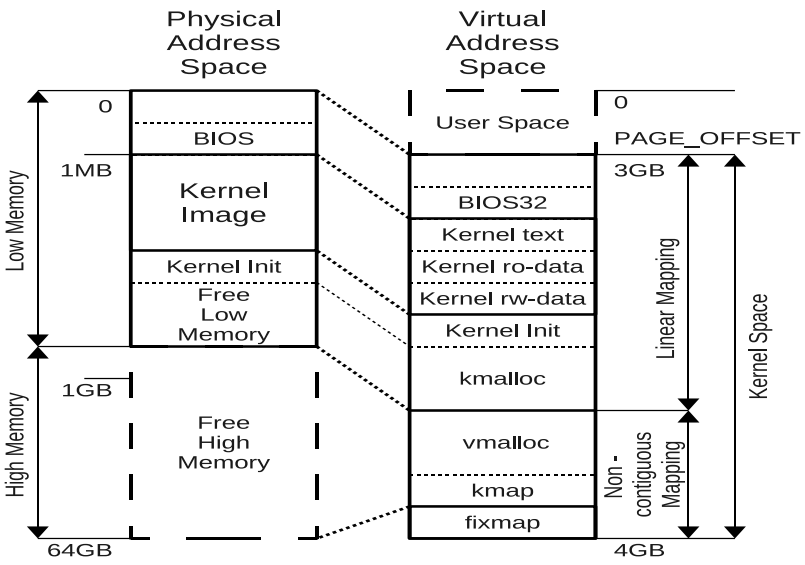


Figure 1: Typical Linux Memory Mapping

these patches impose virtually no performance overhead despite a moderate increase in memory consumption.

The rest of the paper is structured as follows. We start with Linux kernel modeling in Section 2 and present the model checking results in Section 3. Then we present and evaluate our solution in Section 4 and Section 5, respectively. After that, we examine limitations of our solution and suggest possible improvements in Section 6. Finally, we discuss related work in Section 7 and conclude our paper in Section 8.

2. MODELING AND DESIGN

2.1 Murphi Background

Our abstract modeling of Linux kernel memory protection is based on Murphi [10], which is both a language and a tool for model verification with explicit state enumeration. To use Murphi, we need to write a Finite State Machine (FSM) description, which will be taken as an input by Murphi to produce an executable. The executable, once started, will perform the model verification task and produce the output detailing the verification results.

In Murphi, a finite state machine description consists of three parts: a set of *states*, a set of *transition rules* and an *initial state*.

- The set of *states* is implicitly defined through the declaration of global variables. Each combination of values of each variable naturally produces a unique state of the system. Note that not all of these potential combinations are reachable in the FSM.
- The transitions between the states of FSM are defined through a set of *transition rules*, each consisting of two parts, a *guard* and an *action*. A guard is a logic expression that determines the conditions under which an action can be taken. An action is a set of instructions to manipulate the global variables, thus transitioning the FSM from one state to another. An action is performed if, and only if the corresponding guard is evaluated to be *TRUE*.
- An *initial state* is defined by a special rule with an action which is executed only once, right before the FSM state exploration begins. This allows to explicitly define an initial state of the FSM.

For model verification purposes, Murphi also adds to the FSM a fourth component: *invariants*. The invariants are a set of logic expressions which define a set of *safe* states of FSM. A state is only considered *safe* iff all invariants evaluate to *TRUE* in this state. With that, model verification is performed by exploring all reachable states, starting with a given initial state. The state space exploration is performed by applying all possible rules to all states that have already been reached with standard algorithms such as depth- or breadth-first search. For each newly discovered reachable state Murphi evaluates the invariants. Should any invariant evaluate *FALSE*, the system reports an error and prints out a set of states and transitions that led to the unsafe state. The state exploration continues forward only when all invariants evaluate to *TRUE* for the newly discovered state. In this work we apply model checking twice: one in detecting the $W \oplus X$ violation in the current Linux kernel (Section 3) and another in validating our suggested solution (Section 4).

2.2 Linux Kernel Memory Model

For proper modeling, it is important to understand how physical memory is being organized and mapped in Linux. On a typical 32-bit Linux system with the paging-based virtual memory enabled, the 4GB virtual memory space is split (Figure 1) between user space (bottom 3GB) and kernel space (top 1GB). In other words, the kernel space starts at address $PAGE\_OFFSET = 0xc0000000$ ($3GB$) and stretches up to the end of virtual address space $0xffffffff$ ($4GB-1$). We also notice that on an i386 architecture, physical memory is typically split onto two regions: *low memory*

and *high memory*. The low memory region starts at address $0x00000000$ and ends at around $800MB$.[3] In Linux, this region is mapped directly into the kernel space, starting at *PAGE_OFFSET* and called *linear mapping*. Such linear mapping allows for simple translation between the physical and virtual addresses for all pages within this region. Specifically, the virtual address of any location within low memory can be obtained by adding *PAGE_OFFSET* value to the physical address. The reverse translation is performed by simply subtracting *PAGE_OFFSET* from the virtual address, without the need for page table lookup.

We point out that the linear mapping is established at kernel startup and persists all the way through system shutdown [5, 13, 23]. The virtual address space within this region is used to contain the following parts of the Linux kernel: BIOS32 services, kernel text, kernel read-only data, kernel read-write data and dynamic memory allocations that require contiguous physical pages (e.g., through *kmalloc()* call). Kernel initialization routines are also loaded here and later released as free memory.

Assuming the total amount of physical memory installed in the system is greater than the maximum size of low memory for the current kernel configuration, the high memory lays in the physical address space immediately following the low memory. However, unlike low memory, high memory is only mapped into kernel space when needed. There are two basic mechanisms through which this memory is mapped: *vmalloc()* and *kmap()*. The major difference between the two is that the former is used for long-term allocation of non-contiguous physical memory into contiguous virtual address space (e.g., for loadable kernel module allocation), while the latter is used strictly for short-term access to physical pages located in the high memory. Another difference is that vmalloc() may allocate pages from either high or low memory, while kmap() is strictly used for high memory only. Also, when vmalloc() allocates a page from low memory, it creates an *alias*: the same physical page will be mapped into the kernel virtual address space twice, one time in linear area and another one in vmalloc() area. The problem with such aliasing is that memory management subsystem has to ensure consistency of page attributes between all of the aliases. In the context of this work, both vmalloc() and kmap() areas play the same role: mapping non-contiguous physical pages into the kernel address space. Therefore, we treat them as the same in our model.

There is another memory area in Linux kernel called *fixmap*, which is located at the very end of the address space and mainly used by kernel to establish reserved, pre-defined fixed mappings such as PCI control registers and other memory-mapped services. Similar to the linear mapping, the fixmap region is established at kernel startup and persists all the way until system shutdown. Since this area does not represent an unique type of mapping that is distinct from the ones described above, we do not explicitly include it in the model.

Note that all modern multi-tasking operating systems rely on hardware support to provide virtual memory (and the $W \oplus X$ is enforced only when the virtual memory is enabled). In order to accurately model hardware support, we therefore include an abstraction of hardware memory subsystem in our model. Specifically, our model covers the mapping from physical memory to virtual memory as well as the related page tables and access flags. Our model is based on a single-level abstraction of the i386 family paging mechanism [2]. In other words, the mapping of virtual addresses to physical pages is modeled by a flat page table array (*pg_table_t*).[4] Each entry (*pte_t*) of that array corresponds to a virtual page (if *mapped*) and holds related attributes such as physical address (*addr*) it is currently mapped to and access permissions (*prot*). In our model we use *pgprot_t* type for page attribute tracking. This type defines two page access attributes: "write" and "execute". Since kernel pages are not swappable and all mapped pages are always readable, we do not model the "read" and "present" flags. Also, since this model is only concerned with kernel space, we do not model the *user/supervisor* flag. For the physical memory, due to the aliasing, we model them as an array (*frame_table_t*) of physical memory frames, with each physical page (*frame_t*) holding a reference count (*reference_count* or the number of mapped virtual pages) and a most permissive set (*prot*) of the attributes derived from all the virtual aliases pointing to it. As a result, the global definition of the page table array and the physical memory array define the possible *states* in our model.

```
-- page protection attributes
pgprot_t: record
  w: boolean;
  x: boolean;
end;

-- page table entry
-- (mem_index is the memory frame number: [1..
    mem_size])
pte_t: record
  mapped: boolean;
  prot: pgprot_t;
  addr: mem_index;
end;

-- single-level page table
-- (page_index is the page table index: [1..
    pt_size])
pg_table_t: array[page_index] of pte_t;

-- physical memory frames
frame_t: record
    reference_count: 0..pt_size;
    prot: pgprot_t;
end;
frame_table_t: array[mem_index] of frame_t;
```

Based on the above abstraction, our model then captures the specifics of the initial state of Linux kernel. As mentioned earlier and shown in Figure 1, the initialization of Linux kernel memory involves three main parts: kernel linear mapping, static kernel image mapping, and mapping of BIOS32 services. Accordingly, our model represents them by establishing three basic properties as part of the model's *initial state*: ***S1*** *- linear mapping*, ***S2*** *- static kernel mapping*, and ***S3*** *- BIOS32*. The details about them can be found in Appendix A.

After that, we further obtain the transition rules in our model. In particular, based on the Linux kernel source code and our domain knowledge, we identify and extract a number of kernel function routines or application program in-

[3]The exact value depends on the physical memory size and other compile-time and run-time kernel configuration.

[4]Multiple levels of page tables are not necessary as they only allow to map large numbers of pages more efficiently and do not introduce any additional qualities related to this work. The same also stands true for the "large pages" introduced by Page Size Extension (PSE) technology [2].

terfaces (APIs) that are used to affect the kernel memory mapping. Some of them are:

- *map_vm_area()* maps a physical page to a virtual address.
- *static_protections()* ensures that pre-defined areas of kernel's virtual address space always have correct attributes (kernel code should stay executable, kernel data readable and so on.)
- *cpa_process_alias()* checks all mapped aliases for a given physical page frame and updates them as necessary.
- *__change_page_attr_set_clr()* receives a block request for memory attribute change and translates it into a series of attribute and alias check calls for each individual page.
- *__change_page_attr()* executes attribute change for the individual pages.

To represent them, we derive a set of basic rules to capture their behavior especially when they perform memory mapping, re-mapping or change memory page attributes. By doing so, we avoid the need of understanding specific memory use-cases such as kernel module loading or unloading (as it is already captured with these APIs). In our model, we have three key transition rules and use them in our Murphi-based FSM description.

- ***T1*** - *Set*: This transition rule sets W and/or X flags for a given page table entry.
- ***T2*** - *Clear*: This transition rule clears W and/or X flags for a given page table entry.
- ***T3*** - *Map*: This transition rule changes the mapping between a physical frame and a virtual page.

```
ruleset i: page_index do
  ruleset x: boolean do
    ruleset w: boolean do

      -- set W and/or X for virtual page i
      rule "T1: Set"
        true ==> begin
          set_mem_perm(i, 1, w, x);
        end;

      -- clear W and/or X for virtual page i
      rule "T2: Clear"
        true ==> begin
          clr_mem_perm(i, 1, w, x);
        end;

      -- map page i to physical address pa with
          prot attributes
      ruleset pa: mem_index do
        rule "T3: Map"
          true ==> begin
            map_vm_area(i, pa, prot);
         end;
      end;

    end;
  end;
end;
```

3. ANALYSIS

After obtaining the abstract model of Linux kernel, we further define the invariants to analyze possible Linux kernel states. Since our focus in this work is on the $W \oplus X$ enforcement, we established the following properties through invariants in the model:

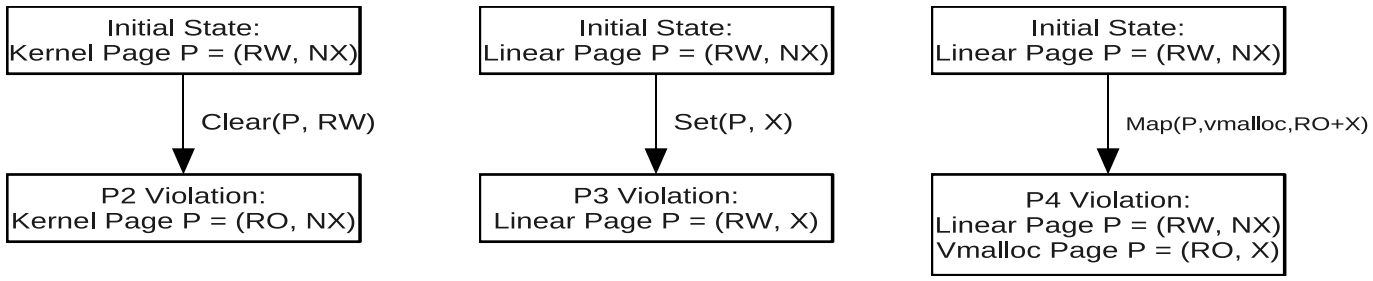
- ***P1***: Kernel code should always be executable and read-only.
- ***P2***: Kernel data should always be non-executable, the read-only kernel data should remain read-only, and read-write kernel data should always be writable.
- ***P3***: No page will be writable and executable at the same time in order to not violate $W \oplus X$.
- ***P4***: All virtual aliases of each physical page should have consistent access permissions.

```
invariant "P1: Kernel ROX Code"
  true -> kernel_code_rox() = true;

invariant "P2: Kernel RO data"
  true -> kernel_rodata_ronx() = true;

invariant "P2: Kernel RW data"
  true -> kernel_rwdata_rwnx() = true;

invariant "P3: W xor X"
  true -> w_and_x() = false;

invariant "P4: Alias consistency"
  true -> page_alias_matching() = true;
```

More specifically, the P1 invariant is necessary to keep kernel code executable because non-executable kernel code will lead to an immediate system crash. The P2 invariant ensures that read-only kernel data that holds constants cannot be modified and that read-write data is always accessible. It also ensures that static kernel data cannot be executed. The P3 invariant states that any given page cannot be writable and executable at the same time. This property is necessary to prevent code injection and is focal point of this work. The P4 invariant is necessary to prevent code injection by accessing an alias which is mapped into a different virtual address with different access permissions.

With invariants in place, the model checker is able to provide us with examples of possible transitions if they lead to an unsafe state that violates $W \oplus X$ policy. Our experience with the model checker indicates that there is no P1 violation in the current Linux design and implementation. However, it reports violations for all other three invariants (Figure 2).

P2 violation The violation of P2 arises when kernel read-write data region is set as read-only. The problem stems from the fact that *static_protections()* does not check for the correctness of new access flags set for the kernel *read-write data* region. While this issue does not directly allow for code injection, setting read-write data (Figure 2(a)) as read-only provides a vector for a denial of service attack. The reason is that the kernel assumes that its read-write data is always writable and is not equipped to handle this situation. This issue has been confirmed by a kernel crash, which immediately follows the call of *set_pages_ro()* for the read-write data region. During the investigation of this issue, we have also identified and confirmed a kernel bug that



(a) P2 Violation: Kernel Read/Write Data (b) P3 Violation: W and X (c) P4 Violation: inconsistent aliases

Figure 2: Property Violations

can be used to apply improper page attributes to a memory region when large (2MB) pages are used in the kernel space (Section 4).

P3 violation The violation of P3 happens in two scenarios. The first one occurs in the initial state. More specifically, the way the BIOS32 is mapped into the kernel space during the initialization directly contradicts the P3 invariant. Note that the BIOS32 services contain executable code and, therefore, should be set as read-only. However, the Linux kernel simply indiscriminately maps the whole BIOS region into the kernel address space as writable and executable. While the actual BIOS code is typically stored in ROM and cannot be overwritten, such mapping still provides an opportunity for data execution or code modification. In a typical page table dump of the latest Linux kernel (version 2.6.33) shown in Figure 3(a), this violation manifests itself as a set of $RW + X$ pages within the range *0xc0000000 - 0xc2000000*.

The second scenario is related to the original memory management interface that allows for pages to be mapped as writable and executable at the same time, thus violating P3 (Figure 2(b)). The source of the problem is the absence of any access permission verification system for memory pages outside the static kernel image in the default memory management interface, such as when a kernel module is being loaded. To confirm this violation, we loaded several modules and inspected the vmalloc() area of the page tables for $W \oplus X$ violations. We found that Linux kernel indeed does not enforce $W \oplus X$ for mappings in the non-contiguous memory region (Figure 3(a)).

P4 violation This violation of P4 happens when a page from *low memory* region is being mapped into vmalloc() area as executable. This would typically happen during the memory allocation for a loadable kernel module on a system that does not have any *high memory* available for allocation. A module loader would use vmalloc() to load all module sections, including code with execute permissions. However, the physical page is already mapped once in the linear mapping space with $RW + NX$ permissions, creating an opportunity for code injection. We were able to re-create this behavior consistently by loading Linux kernel in a virtual machine with only 128MB of RAM, forcing Linux to allocate all pages from low memory and thus creating aliases for each of vmalloc() allocations.

4. IMPROVEMENTS AND PROTOTYPE

Based on the findings described in the previous section, we propose a few improvements to the Linux kernel. Our improvement involves the changes to the initial states (S1, S2, and S3) as well as the transition rules (T1, T2, and T3) such that no unsafe states will be reached from the revised FSM description. For debugging and verification purposes, we have developed five self-contained and independent patches (in total 703 source code of lines) to the Linux kernel, four of which have been submitted to LKML [17, 18, 19, 20] and are currently in the process of being integrated into the mainline kernel. The remaining one is still being assessed for suitability in the mainline kernel.

In the development of these patches, based on the counter-examples reported from Murphi, we revised the memory management subsystem in Linux. Interestingly, we found that there are two distinct levels of abstraction interfaces to manage Linux kernel memory. The low-level interface is primarily tasked with direct manipulation of page table entries, and the high-level interface is used to abstract page table layout and provide functions like *set_memory_nx()*. At first glance, the low-level interface seems a better target to address previous violations. However, the following significant shortcomings in its design complicate such approach. Therefore, we chose the high-level interface as a target for our solution.

- First, the low-level functions (e.g., *pte_mkexec()*) are intended as simple data manipulation primitives which cannot fail. Our experience indicates that any deviation from this assumption would yield unpredictable results, as the current kernel is not yet designed to handle such failures gracefully.
- Second, these functions operate on instances of *pte_t* type, which by itself does not guarantee that the instance is in fact part of active or future page tables. This, in turn, means that we would either need to check the address of each *pte_t* in question, or enforce $W \oplus X$ on all instances of *pte_t*. Neither of these options is desirable: the former creates a significant performance overhead while the latter one introduces unwanted side effects into all intermediate transformations of *pte_t* instances.

4.1 Fixing P2 Violation: Preserving the *write* access on kernel read-write data

Our first patch fixes the violation of P2 invariant. More specifically, as discussed earlier, this problem is caused by the inability of *static_protections()* to preserve the *write* access to kernel *read-write data* (including the *BSS* section). The fix is a straightforward one. However, in the process

```
---[ Kernel Mapping ]---
0xc0000000-0xc0200000          2M     RW             GLB x  pte
0xc0200000-0xc0600000          4M     ro         PSE GLB x  pmd
0xc0600000-0xc0843000       2316K     ro             GLB x  pte
0xc0843000-0xc0a00000       1780K     RW             GLB x  pte
0xc0a00000-0xf7800000        878M     RW         PSE GLB NX pmd
0xf7800000-0xf79fe000       2040K     RW             GLB NX pte
0xf79fe000-0xf7a00000          8K                           pte
0xf7a00000-0xf8000000          6M                           pmd
0xf8000000-0xf81fe000       2040K                           pte
---[ vmalloc() Area ]---
0xf81fe000-0xf81ff000          4K     RW     PCD     GLB NX pte
0xf81ff000-0xf8200000          4K                           pte
                            [ . . . ]
0xf8247000-0xf824a000         12K     RW             GLB x  pte
0xf824a000-0xf824c000          8K                           pte
0xf824c000-0xf824d000          4K     RW             GLB x  pte
0xf824d000-0xf824f000          8K                           pte
0xf824f000-0xf8274000        148K     RW             GLB NX pte
0xf8274000-0xf8276000          8K                           pte
0xf8276000-0xf8278000          8K     RW             GLB x  pte
0xf8278000-0xf827a000          8K                           pte
0xf827a000-0xf827b000          4K     RW             GLB x  pte
0xf827b000-0xf827d000          8K                           pte
0xf827d000-0xf8296000        100K     RW             GLB NX pte
0xf8296000-0xf8298000          8K                           pte
                          [ . . . ]
```

(a) Vanilla Kernel

```
---[ Kernel Mapping ]---
0xc0000000-0xc00fb000       1004K     RW             GLB NX pte
0xc00fb000-0xc00fd000          8K     ro             GLB x  pte
0xc00fd000-0xc0200000       1036K     RW             GLB NX pte
0xc0200000-0xc0600000          4M     ro         PSE GLB x  pmd
0xc0600000-0xc068e000        568K     ro             GLB x  pte
0xc068e000-0xc0844000       1752K     ro             GLB NX pte
0xc0844000-0xc0a00000       1776K     RW             GLB NX pte
0xc0a00000-0xf7800000        878M     RW         PSE GLB NX pmd
0xf7800000-0xf79fe000       2040K     RW             GLB NX pte
0xf79fe000-0xf7a00000          8K                           pte
0xf7a00000-0xf8000000          6M                           pmd
0xf8000000-0xf81fe000       2040K                           pte
---[ vmalloc() Area ]---
0xf81fe000-0xf81ff000          4K     RW     PCD     GLB NX pte
0xf81ff000-0xf8200000          4K                           pte
                            [ . . . ]
0xf821a000-0xf821d000         12K     ro             GLB x  pte
0xf821d000-0xf821e000          4K     ro             GLB NX pte
0xf821e000-0xf8220000          8K     RW             GLB NX pte
0xf8220000-0xf8222000          8K                           pte
0xf8222000-0xf8223000          4K     RW     PCD     GLB NX pte
0xf8223000-0xf8227000         16K                           pte
0xf8227000-0xf8228000          4K     ro             GLB x  pte
0xf8228000-0xf8229000          4K     ro             GLB NX pte
0xf8229000-0xf822b000          8K     RW             GLB NX pte
                          [ . . . ]
```

(b) Patched Kernel

Figure 3: Dumping Kernel Page Tables (kernel version 2.6.33**)**

of re-mapping our model back to the original Linux kernel source code, we discovered another implementation bug in the function routine *try_preserve_large_pages()*. Specifically, this function incorrectly processes access permission change requests for areas that start on a boundary of a large page (i.e., $2M$), but are smaller than the page itself. This leads to a possibility of setting improper access flags to the memory area located within the same large page, but immediately after the requested one. Specifically, this problem manifested itself by kernel read-write data becoming read-only simply because it was initially mapped within the same large page as kernel's read-only data. Accordingly, we propose two changes in the patch (that affects the file *arch/x86/mm/pageattr.c*): one is to allow *static_protections()* to preserve the *write* access for kernel's read-write data area and another one is to verify each small page within the large page for access flag compatibility [17].

4.2 Fixing P3 Violation: Removing mixed pages in kernel space

Our next three patches address P3 violation, namely the presence of mixed code and data pages in kernel space. Based on our model checking results, our investigation maps the related transition rules that lead to an unsafe state back to the involved kernel routines. By doing so, we are able to identify three distinct sources: BIOS32, loadable kernel modules (LKMs), and static kernel image management.

BIOS32 As discussed in Section 3, the current Linux kernel improperly maps the entire BIOS area into the kernel space as $RW + X$. (Note the BIOS code itself is typically located in read-only memory or ROM.) To resolve this issue, we implemented a patch that dynamically maps BIOS32 services into the kernel space. Based on related BIOS32 documents [4] and [26], it requires at most two pages to be executable per BIOS32 service, and none of them are expected to be writable. As such, a dynamic service mapping of BIOS32 services can be established at the time of service discovery, and with appropriate (and $W \oplus X$-compliant) access permissions. As part of our patch, we added the BIOS32 service mapper to *pci/pcbios.c* and removed unnecessary protection for the area of physical memory under 2MB from *arch/x86/mm/pageattr.c*. Also, we revised the file *mm/init_32.c* to properly report kernel text addresses [18] because of the BIOS32 changes.

LKMs The second source of violating $W \oplus X$ is located in the support of LKMs. In particular, since all vmalloc() allocations default to "data" access mode ($RW + NX$), the only source of mixed pages in this area is LKMs as their code is explicitly marked as "executable". More specifically, dynamic kernel linker allocates each loadable module in two parts: module init and module core. The init part of the module will be discarded after initialization, while core will stay resident in the kernel. In order to minimize a module's footprint, the linker chooses the minimum amount of spacing necessary between each of the module's sections - just enough to accommodate necessary section alignment, which introduces mixed kernel pages. Accordingly, our patch allocates module sections in three groups: text, read-only data, read-write data and further adjusts the linker to align each of the groups on a page boundary, ensuring that each page contains only sections from the same group. Next, our patch assigns a set of appropriate access permissions to all pages of each group as follows: read-only for text and read-only data, non-executable for read-only data and read-write data. As this patch will inevitably introduce additional memory consumption (Section 5), we create a compile-time option (i.e., *CONFIG_DEBUG_SET_MODULE_RONX*) to turn on or off the functionality of this patch as needed [20].

Static kernel image Similar to the support of LKMs, our next patch includes the code to split all sections of the static kernel image into three groups: text, read-only data, and read-write data. Specifically, our patch addresses necessary group alignment by modifying the related linker script (*kernel/vmlinux.lds.S*) and assigns proper access permissions to the pages of each group at the end of kernel initialization (*mm/init.c* [19]). The functionality of this patch is always enabled.

4.3 Fixing P4 Violation: Disallowing memory aliasing with permission conflicts

The remaining patch addresses the P4 violation. In particular, as we have mentioned in Section 2.2, memory alias han-

dling is being implemented in the kernel in *cpa_process_alias()* function. However, the way it handles page aliases specifically excludes the case of an NX update.

```
/* No alias checking for _NX bit modifications */
checkalias = (pgprot_val(mask_set) | pgprot_val(
    mask_clr)) != _PAGE_NX;
```

To enforce $W \oplus X$, our patch needs to modify the alias handling routine such that it will propagate changes of all access flags to all aliases. This can be achieved by setting *checkalias = 1* for all page attribute modifications.

```
static inline pgprot_t
  process_WxorX_violation(pgprot_t prot,
     unsigned long address, unsigned long pfn)
{
  /*
   * We can Oops or Panic here if needed. But for
        now we just print out an error message.
   */
  printk(KERN_ERR "(W xor X) violation: " "VA=0x%
      lx, PFN=0x%lx", address, pfn);
  /* Set NX, just in case */
  pgprot_val(prot) |= _PAGE_NX;
  return prot;
}
```

In addition, our patch implemented a helper routine *process_WxorX_violation()* (a part of *mm/pageattr.c*) for the strict enforcement of $W \oplus X$ property. Particularly, this function will be called for each page that is about to violate $W \oplus X$ property, before the new protection attributes can be applied. Because of the presence of aliasing, this routine is called for any inconsistency of memory protection attributes in aliases as well. If there are conflicting attributes, this function will by default set NX flag for all related pages and logs an error message detailing the associated virtual and physical addresses.

5. EVALUATION

Our improvements aim to make the Linux kernel conform to the $W \oplus X$ property by guaranteeing exclusivity of write and execute page access. This means that all of the kernel code that follows the interfaces in place will automatically be compliant with $W \oplus X$ without additional modifications. To rigorously verify the $W \oplus X$ compliance, we revise the previous FSM description to reflect our patches (Section 4). Also, in order to avoid unnecessary state explosion, we chose a minimal configuration where we only modeled a system that contains one virtual page of each type: kernel text, kernel read-only data, kernel read-write data, linear mapping. To allow for variability, we use two pages in non-contiguous mappings, and model the physical memory with one more page frame than the total size of the virtual address space. In addition, the model contained a proposed memory interface, a full set of rules, and invariants to establish and monitor the $W \oplus X$ property. The model checker examined 27942 states and 7823760 rules without detecting any violations.

In order to further confirm the validity of our approach in a real system, we used a minimal Ubuntu Server 8.04.4 LTS [32] system that runs the latest vanilla Linux 2.6.33 kernel [16]. Note the vanilla Linux kernel has been compiled with Generic Ubuntu configuration, and then booted and inspected. In Figure 3(a), we show the virtual memory layout in the vanilla Linux kernel. It shows that while all necessary elements of code and data separation are indeed present, they are not applied in a consistent manner. Specifically, RO and NX flags are used sparsely. As shown from the detailed kernel page table dump (Figure 3(a)), it fails to establish $W \oplus X$ property.

In comparison, we applied our patch set to the same kernel and repeated the inspection. A clear difference can be observed on Figure 3(b): we have successfully eliminated all pages with mixed access. Specifically, the following changes are noteworthy:

- a 2Mb area between *0xc0000000* and *0xc0200000* is now marked as $RW + NX$, except for two $RO + X$ pages *0xc00fb000* - *0xc00fd000* reserved for BIOS32 services.
- Static Kernel image (*0xc02000000* - *0xc08a3000*) is clearly partitioned in three sections: $RO + X$ code (*0xc02000000* - *0xc068e000*), $RO+NX$ read-only data (*0xc08a3000* - *0xc0844000*), and $RW+NX$ read-write data (*0xc0844000* - *0xc08a3000*).
- Loadable Kernel Modules (see addresses *0xf821a000* - *0xf8220000* and *0xf8227000* - *0xf822b000*) are now clearly split into three parts: $RO+X$ code, $RO+NX$ read-only data, and $RW + NX$ read-write data.

Performance and Memory Overhead To evaluate the performance overhead introduced, we measured its runtime overhead with three tasks: UnixBench 5.1.2 [31] (index test group, SMP and Uniprocessor configurations), Linux kernel compilation, and compression time of 10GB random data stream. All tests have been performed on Ubuntu Server 8.04.4 LTS with Linux 2.6.33 compiled for i386 architecture in standard Ubuntu Server configuration except for PAE enabled, and XEN disabled. Our test platform is a Gigabyte MA78G-DS3HP system (AMD RS780/SB700 chipset) with dual-core AMD Athlon 4850e processor (family 15, model 107, stepping 2) and 8GB of PC2-6400 RAM in dual-channel configuration (4x2GB, CL5). The results are shown in Table 1.

Our results indicate that our patch set does not affect overall system performance in a measurable way. In particular, the first four patches (related to static kernel image, LKM, and BIOS32) do not introduce any additional performance overhead as all additional work are mainly performed at compile-time. Though there is a slight overhead incurred during the kernel/module initialization, it does not affect the runtime performance after initialization. Among the five patches, the only patch that involves run-time penalty is the compliance checking of $W \oplus X$ for each kernel page table update (i.e., in the helper routine *process_WxorX_violation()*).

Next, we evaluate the memory overhead introduced into the kernel by our patches. We first investigate the difference in the size of the static kernel image and memory allocated to individual modules. As can be seen on Figure 4, the size of the static kernel image has increased from $6793KB$ ($4660KB$ text + $2133KB$ data) to $6796KB$ ($4664KB$ text + $2132KB$ data). This increase constitutes a mere 0.04%, and is thus not significant.

We also used the same system to load 44 kernel modules of varying sizes. Due to the fact that our patch set affects the layout of different module sections, the total size of these 44 modules (reported by running the *lsmod* command) is increased from $1,265,502B$ to $1,493,138B$ (an increase of 22.01%). Note that the module sizes reported by *lsmod*

(a) Vanilla Kernel

(b) Patched Kernel

Figure 4: Kernel Virtual Memory Layout

Benchmark	Vanilla	Patched	Overhead%
UnixBench UP (index)	737.88	741.8	0.53%
UnixBench SMP (index)	1317.13	1310.88	-0.47%
Kernel Compilation (seconds)	1712	1725	0.76%
Gzip test (seconds)	2890	2873	-0.59%

Table 1: Run-time $W \oplus X$ enforcement overhead

could be misleading, as they do *not* reflect the fact that kernel allocates memory for modules at the page granularity. This means that the true measure of module memory consumption should be the memory (in terms of pages) allocated to the module, not the module size. With that, if we can take the whole-page allocation into account, the memory overhead is increased from 322 pages (size of $4K$) to 382 pages (an increase of 22.17%). We believe such memory overhead is moderate and within an acceptable range for modern server and desktop systems, especially considering the current price drop of physical memory. In the meantime, we also recognize that such increase in memory consumption might be unwanted in certain memory-constrained applications, such as embedded systems. Because of that, our patch is provided as a compile-time option that may be enabled or disabled as needed.

6. DISCUSSION

As with many other real-world protection systems, our approach comes with a few limitations. First of all, for the purpose of verifying Linux kernel $W \oplus X$ enforcement, our approach assumes that the static kernel image and LKMs are trusted. More specifically, the kernel (including LKMs) is assumed to follow the transition rules (Section 2.2) to manage the kernel memory. If this assumption is violated, the invariants derived in this work may not be valid. Note that such trust can be potentially established by means of kernel/driver signing [12, 27, 28, 30], which falls outside the scope of this work. Second, it is important to note that while providing $W \oplus X$ is helpful to block code injection attacks, $W \oplus X$ itself does not prevent other types of attacks, such as "return-into-libc" [14]. Third, since our modeling is based on the correctness of the internal kernel APIs, any code inside the function that modifies page tables directly will not be prevented from doing so. Finally, although we establish a $W \oplus X$ property in the Linux kernel, there are a few exceptions that we had to consider for the implementation of our patches. For example, a special accommodation had to be made for kernel's built-in function tracing facility - *ftrace* [1]. The function tracing facility essentially requires dynamically modifying kernel code, which is in conflict with our $W \oplus X$ enforcement. As a result, we have to allow a short window of $W \oplus X$ violation while ftrace is in use to place its trace points (that requires modifying kernel's code at run time).

7. RELATED WORK

Model checking for improved security The first category of related work includes recent efforts that leverage model checking to improve systems security. For example, Mitchell et al. [22] applied model checking to successfully verify the correctness of (and find bugs in) security protocol specifications. Chen et al. [8] utilized model checking to analyze or demystify the confusing setuid system calls. Others [6, 7, 11, 29, 34] have used software model checking and static analysis to find a general class of bugs in source code. In contrast, our focus is on the analysis and verification of $W \oplus X$ enforcement for Linux kernel memory protection.

$W \oplus X$ enforcement The second category of related work aims at enforcing $W \oplus X$ as an effective defense against code injection attacks. For example, SecVisor [30] and NICKLE [28] use custom hypervisors to enforce code protection and data non-execution. Such protection is achieved through effectively separating code and data address spaces. Both methods make it impossible to modify code and to execute data without going through the special authentication mechanisms controlled by a hypervisor. Note that even with hardware-based full virtualization support, they inevitably lead to significant performance degradation, as policy enforcement requires additional management activities that consume extra clock cycles and cause cache pollution. Others take advantage of standard hardware protection features in the most straight-forward manner and introduce minimal impact. For examples, both PaX [25] and ExecShield [33] make use of standard memory protection features found in most architectures to achieve $W \oplus X$ property. Shared with our approach, these patches separate memory pages into two categories: code and data. Code pages are set as $RO + X$, while data as $RW + NX$. However, our approach is different from them in that we use a model checking approach to systematically analyze the $W \oplus X$ protection in the Linux kernel memory space while others mainly concentrate on userspace application protection. Also, our solution is based on the model-checking approach, which is appropriate for formal verification.

Other code injection defense mechanisms The third category of related work contains other approaches to defend against code injection attacks. For example, two other notable ways in this category include Address Space Layout Randomization (ASLR) [24, 25] and Instruction Set Randomization [15, 3]. ASLR is based on the idea of randomization of all major components within the application address space. This typically involves introducing random offsets in the layout of all major sections of the primary executable and the libraries it requires at link-time. This type of protection is already included in the mainline kernel and used exclusively for userspace. Instruction set randomization takes a somewhat different approach by randomizing the actual machine instruction set. This is achieved by creating a virtual machine with unique instruction encoding for each run. Compatibility with pre-compiled binaries is established by load-time binary translator, which converts the code from well-known "generic" instruction encoding to the encoding used in the specific virtual machine. Such approaches are not widely deployable since dynamic instruction sets are not supported by any modern hardware and software-based emulation likely introduces prohibitive performance overhead.

8. CONCLUSION

In this paper, we have presented a model checking-based approach to analyze the $W \oplus X$ protection in the Linux kernel space. Our modeling has led to the discovery of several real problems in the current Linux kernel design and implementation. Based on the model checking results, we have accordingly developed five kernel patches to fix them and four of them are in the process of being integrated into the mainline Linux kernel. Our evaluation with these patches indicate that they involve minimal changes and incur negligible performance overhead to the Linux kernel.

Acknowledgments The authors would like to thank the anonymous reviewers for their numerous, insightful comments that greatly helped improve the presentation of this paper. This work was supported in part by the US Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI), the US Air Force Research Laboratory (AFRL) under contract FA8750-09-1-0224, and the US National Science Foundation (NSF) under Grants 0852131, 0855297, 0855036, 0910767, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ARO, the AFRL, and the NSF.

9. REFERENCES

[1] A Look at ftrace. `http://lwn.net/Articles/322666/`.

[2] Advanced Micro Devices. *AMD64 Architecture Programmerąŕs Manual Volume 2: System Programming*, 3.14 edition, September 2007.

[3] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *CCS '03: Proceedings of the 10th ACM Computer and Communications Security Conference*, 2003.

[4] T. C. Block. *Standard BIOS 32-bit Service Directory Proposal*. Phoenix Technologies Ltd., 0.4 edition, June 1993.

[5] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, third edition, 2005.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI'08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2008.

[7] H. Chen and D. Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. In *CCS'02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.

[8] H. Chen, D. Wagner, and D. Dean. Setuid Demystified. In *Security '02: Proceedings of the 11th Conference on USENIX Security Symposium*, 2002.

[9] I. Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. Intel Corp., 2006. Publication number 253668.

[10] D. L. Dill. The Murphi Verification System. `http://eprints.kfupm.edu.sa/70602/1/70602.pdf`, 1996.

[11] J. Franklin, A. Seshadri, N. Qu, S. Chaki, and A. Datta. Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor Research Area. Technical report, June 2008.

[12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *SOSP '03: Proceedings of the 19th Symposium on Operating System Principles*, October 2003.

[13] M. Gorman. *Understanding The Linux Virtual Memory Manager*. Prentice Hall, May 2004.

[14] R. Hund, T. Holz, and F. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Security '09: Proceedings of the 18th Conference on USENIX Security Symposium*. USENIX Association, 2009.

[15] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *CCS '03: Proceedings of the 10th ACM Computer and Communications Security Conference*, 2003.

[16] The Linux Kernel Archives. `http://www.kernel.org`.

[17] S. Liakh and X. Jiang. [1/4,tip:x86/mm] correcting improper large page preservation. https://patchwork.kernel.org/patch/90045/, 2010.

[18] S. Liakh and X. Jiang. [2/4,tip:x86/mm] set first mb as rw+nx. https://patchwork.kernel.org/patch/90048/, 2010.

[19] S. Liakh and X. Jiang. [3/4,tip:x86/mm] nx protection for kernel data. https://patchwork.kernel.org/patch/90046/, 2010.

[20] S. Liakh and X. Jiang. [4/4,tip:x86/mm] ro/nx protection for loadable kernel modules. https://patchwork.kernel.org/patch/90047/, 2010.

[21] E. P. M. Michalis Polychronakis, Kostas G. Anagnostakis. An Empirical Study of Real-world Polymorphic Code Injection Attacks. In *LEET '09: Proceedings of the 2nd USENIX Workshop on*

Large-Scale Exploits and Emergent Threats. Usenix Association, April 2009.

[22] J. Mitchell, V. Shmatikov, and U. Stern. Finite-State Analysis of SSL 3.0. In *Security '98: Proceedings of the 7th Conference on USENIX Security Symposium*, 1998.

[23] A. Nayani, M. Gorman, and R. S. de Castro. Memory Management in Linux - Desktop Companion to the Linux Source Code. `http://www.ecsl.cs.sunysb.edu/elibrary/linux/mm/mm.pdf`, May 2002.

[24] PaX ASLR. `http://pax.grsecurity.net/docs/aslr.txt`.

[25] PaX NOEXEC. `http://pax.grsecurity.net/docs/noexec.txt`.

[26] PCI Special Interest Group. *PCI BIOS Specification*, 2.1 edition, August 1994.

[27] S. Pearson. Trusted Computing Platforms, the Next Security Solution. Technical report, HP Laboratories, November 2002.

[28] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*. Springer-Verlag, 2008.

[29] B. Schwarz, H. Chen, D. Wagner, G. Morrison, J. West, J. Lin, and W. Tu. Model Checking An Entire Linux Distribution for Security Violations. In *ACSAC'05: Proceedings of the 21st Annual Computer Security Applications Conference*, 2005.

[30] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of the 21st ACM SIGOPS symposium on Operating systems principles*. ACM, 2007.

[31] J. Tombs, B. Smith, R. Grehan, T. Yager, D. C. Niemi, and I. Smith. Unixbench-5.1.2. `http://code.google.com/p/byte-unixbench/`.

[32] Ubuntu. `http://www.ubuntu.com/`.

[33] A. van de Ven. Limiting Buffer Overflows with ExecShield. *Red Hat Magazine*, July 2005.

[34] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI'04: Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.

Appendix A: Defining the Initial State in the Model

```
procedure map_linear();
begin
 -- page count starts from 1, therefore +1
 for va: (page_offset+1) ..
       (page_offset+kernel_direct_size) do
    -- I1: linear kernel mapping
    pt[va].addr := va - page_offset;
    pt[va].present := true;
 end;
end;

procedure map_static_kernel();
begin
 -- page count starts from 1, therefore +1
 for va: (page_offset+1) ..
       (page_offset+kernel_direct_size) do

    -- I2: are we mapping code and ro-data?
    if(va <= (page_offset+kernel_text_size
              + kernel_ro_size))
    then
      -- yes, set page as R/O
      pt[va].prot.w := false;
    else
      -- no, set it as writable
      pt[va].prot.w := true;
    end;

    -- I2: are we mapping kernel data?
    if(va>(page_offset+kernel_text_size))
    then
  -- yes, set it non-executable
  pt[va].prot.x := false;
    else
      -- no, set code as executable
      pt[va].prot.x := true;
    end;
  end;
end;

procedure map_bios();
begin
 -- page count starts from 1, therefore +1
 for va: bios_start .. bios_end do
    -- I3: BIOS mapping
    pt[va].addr := va - page_offset;
    pt[va].present := true;
    pt[va].prot.w := true;
    pt[va].prot.x := true;
 end;
end;

startstate
begin
        clear pt;
        map_linear();
        map_static_kernel();
        map_bios();
end;
```