An Architectural Approach to Preventing Code Injection Attacks

Ryan Riley
Purdue University
rileyrd@cs.purdue.edu

Xuxian Jiang
George Mason University
xjiang@gmu.edu

Dongyan Xu
Purdue University
dxu@cs.purdue.edu

Abstract

Code injection attacks, despite being well researched, continue to be a problem today. Modern architectural solutions such as the NX-bit and PaX have been useful in limiting the attacks, however they enforce program layout restrictions and can often times still be circumvented by a determined attacker. We propose a change to the memory architecture of modern processors that addresses the code injection problem at its very root by virtually splitting memory into code memory and data memory such that a processor will never be able to fetch injected code for execution. This virtual split memory system can be implemented as a software only patch to an operating system, and can be used to supplement existing schemes for improved protection. Our experimental results show the system is effective in preventing a wide range of code injection attacks while incurring acceptable overhead.

Keywords: *Code Injection, Secure Memory Architecture*

1. Introduction

Despite years of research, code injection attacks continue to be a problem today. Systems continue to be vulnerable to the traditional attacks, and attackers continue to find new ways around existing protection mechanisms in order to execute their injected code. Code injection attacks and their prevention has become an arms race with no obvious end in site.

A code injection attack is a method whereby an attacker inserts malicious code into a running process and transfers execution to his malicious code. In this way he can gain control of a running process, causing it to spawn other processes, modify system files, etc. If the program runs at a privilege level higher than that of the attacker, he has essentially escalated his access level. (Or, if he has no privileges on a system, then he has gained some.)

A number of solutions exist that handle the code injection problem on some level or another. Architectural approaches [1, 2, 3] attempt to prevent malicious code execution by making certain pages of memory non-executable. This protection methodology is effective for many of the traditional attacks, however attackers still manage to circumvent them [4]. In addition, these schemes enforce specific rules for program layout with regards to separating code and data, and as such are unable to protect memory pages that contain *both*. Compiler based protection mechanisms [5, 6, 7] are designed to protect crucial memory locations such as function pointers or the return address and detect when they have been modified. These methods, while effective for a variety of attacks, do not provide broad enough coverage to handle a great many modern vulnerabilities [8]. Both of these techniques, architectural and compiler based, focus on preventing an attacker from executing his injected code, *but do nothing to prevent him from injecting and fetching it in the first place*.

The core of the code injection problem is that modern computers implement a von Neumann memory architecture [9]; that is, they use a memory architecture wherein code and data are both accessible within the same address space. This property of modern computers is what allows an attacker to inject his attack code into a program as data and then later execute it as code. Wurster et al [10] proposed a technique to defeat software self checksumming by changing this property of modern computers (and hence producing a Harvard architecture [11, 12]), and inspired us to consider the implications such a change would have on code injection.

We propose virtualizing a Harvard architecture on top of the existing memory architecture of modern computers so as to prevent the injection of malicious code entirely. A Harvard architecture is simply one wherein code and data are stored separately. Data cannot be loaded as code and vice-versa. In essence, we create an environment wherein any code injected by an attacker into a process' address space cannot even be addressed by the processor for execution. In this way, we are attacking the code injection problem at its root by regarding the injected malicious code as data and making it unaddressable to the processor during an instruction fetch. The technique can be implemented as a software only patch for the operating system, and our implementation for the x86 incurs a very reasonable performance penalty, on average between 10 and 20%. Such a software only technique is possible through careful exploitation of the two translation lookaside buffers (TLBs) on the x86 architecture in order to split memory in such a way that it enforces a strict separation of code and data memory.

2. Related Work and Motivation

Research on code injection attacks has been ongoing for a number of years now, and a large number of protection methods have been researched and tested. There are two classes of techniques that have become widely supported in modern hardware and operating systems; one is concerned with preventing the execution of malicious code after control flow hijacking, while the other is concerned with preventing an attacker from hijacking control flow.

The first class of technique is concerned with preventing an attacker from executing injected code using non-executable memory pages, but does not prevent the attacker from impacting program control flow. This protection comes in the form of hardware support or a software only patch. Hardware support has been put forth by both Intel and AMD that extends the page-level protections of the virtual memory subsystem to allow for non-executable pages. (Intel refers to this as the "execute-disable bit" [3].) The usage of this technique is fairly simple: Program information is separated into code pages and data pages. The data pages (stack, heap, bss, etc) are all marked non-executable. At the same time, code pages are all marked read-only. In the event an attacker exploits a vulnerability to inject code, it is guaranteed to be injected on a page that is non-executable and therefore the injected code is never run. Microsoft makes use of this protection mechanism in its latest operating systems, calling the feature Data Execution Protection (DEP) [1]. This mediation method is very effective for traditional code injection attacks, however it requires hardware support in order to be of use. Legacy x86 hardware does not support this feature. This technique is also available as a software-only patch to the operating system that allows it to simulate the execute-disable bit through careful mediation of certain memory accesses. PAX PAGE-EXEC [2] is an open source implementation of this technique that is applied to the Linux kernel. It functions identically to the hardware supported version, however it also supports legacy x86 hardware due to being a software only patch.

The second class of technique has a goal of preventing the attacker from hijacking program flow, but does not concern itself with the injected code. Works such as Stack-Guard [5] accomplish this goal by emitting a "canary" value onto the stack that can help detect a buffer overflow. ProPolice [6] (currently included in gcc) builds on this idea by also rearranging variables to prevent overflowed arrays from accessing critical items such as function pointers or the return address. Stack Shield [7] uses a separate stack for return addresses as well as adding sanity checking to `ret` and function pointer targets. Due to the fact that these techniques only make it their goal to prevent control flow hijacking, they tend to only work against known hijacking techniques. That means that while they are effective in some cases, they may miss many of the more complicated attacks. Wilander et al [8], for example, found that these techniques missed a fairly large percentage (45% in the best case) of attacks that they implemented in their buffer overflow benchmark.

Due to the fact that the stack based approaches above do not account for a variety of attacks, in this work we are primarily concerned with addressing limitations in the architectural support of the execute-disable bit. While this technique is widely deployed and has proven to be effective, it has limitations. First, programs must adhere to the "code and data are always separated" model. In the event a program has pages containing both code and data the protection scheme cannot be used. In fact, such "mixed pages" do exist in real-world software systems. For example, the Linux kernel uses mixed pages for both signal handling [13] as well as loadable kernel modules. A second problem with these schemes is that a crafty attacker can disable or bypass the protection bit using library code already in the process' address space and from there execute the injected code. Such an attack has been demonstrated for the Windows platform by injecting code into non-executable space and then using a well crafted stack containing a series of system calls or library functions to cause the system to create a new, executable memory space, copy the injected code into it, and then transfer control to it. One such example has been shown in [4].

It is these two limitations in existing page-level protection schemes (the forced code and data separation and the bypass methodology) that provide the motivation for our work, which architecturally addresses the code injection problem at its core. Note that our architectural approach is orthogonal to research efforts on system randomization, such as Address Space Layout Randomization (ASLR) [14, 15, 16, 17] and Instruction Set Randomization (ISR) [18, 19, 20]. We are also distinct from other work that focuses specifically on preventing array overflow using a compiler or hardware, such as [21]. We point out that these alternate systems all work on a single memory architecture wherein code and data are accessible within the same address space. Our approach, to be described in the next section, instead creates a different memory architecture where code and data are separated.

3. An Architectural Approach

At its root, code injection is a problem because processors permit code and data to share the same memory address space. As a result, an attacker can inject his payload as data and later execute it as code. The underlying assumption relied on by attackers is that the line between code and data is blurred and not enforced. For this reason, we turn to an alternative memory architecture that does not permit code and data to be interchanged at runtime.

3.1. The Harvard and von Neumann Memory Architectures

Modern computers and operating systems tend to use what is known as a von Neumann memory architecture [9]. Under a von Neumann system there is one physical memory which is shared by both code and data. As a consequence of this, code can be read and written like data and data can be executed like code. Many systems will use segmentation or paging to help separate code and data from each other or from other processes, but code and data end up sharing the same address space. Figure 1a illustrates a von Neumann architecture.

An architecture not found in most modern computers (but found in some embedded devices or operating systems, such as VxWorks [22]) is known as a Harvard architecture [11, 12]. Under the Harvard architecture code and data each have its own physical address space. One can think of a Harvard architecture as being a machine with two different physical memories, one for code and another for data. Figure 1b shows a Harvard architecture.

3.2. Harvard and Code Injection

A code injection attack can be thought of as being carried out in four distinct, but related, stages:

1. The attacker injects code into a process' address space.
2. The attacker determines the address of the injected code.
3. The attacker somehow hijacks the program counter to point to the injected code.
4. The injected code is executed.

The mediation methods mentioned in section 2 are designed to handle the problem by preventing either step 3 or 4. Non-executable pages are designed to prevent step 4, while compiler based approaches are meant to prevent step 3. In both cases, however, the malicious code is injected, but execution is somehow prevented. Our solution, on the other hand, effectively stops the attack at step 1 by preventing the successful injection of the malicious code into a process' *code space*. (The purist will note that in the implementation method described in section 4 the attack is not technically stopped until step 4, however the general approach described here handles it at step 1.)

The Harvard architecture's split memory model makes it suitable for the prevention of code injection attacks due to the fact that a strict separation between code and data is enforced at the hardware level. Any and all data, regardless of the source, is stored in a different physical memory from instructions. Instructions cannot be addressed as data, and data cannot be addressed as instructions. This means that in a Harvard architecture based computer, a traditional code injection attack is not possible because the architecture is not capable of supporting it after a process is initially setup. The attacker is simply unable to inject any information whatsoever into the instruction memory's address space and at the same time is unable to execute any code placed in the data memory. The architecture simply does not have the "features" required for a successful code injection attack. However, we point out that this does not prevent an attacker from mounting non control injection attacks (e.g., non-control-data attack [23]) on a Harvard architecture. We touch on these attacks in section 6.

3.3. Challenges in Using a Harvard Architecture

While a Harvard architecture may be effective at mitigating code injection, the truth of the matter is that for any new code injection prevention technique to be practical it must be usable on modern commodity hardware. As such, the challenge is to construct a Harvard architecture on top of a widely deployed processor such as the x86. We first present a few possible methods for creating this Harvard architecture on top of the x86.

Modifying x86

One technique for creating such an architecture is to make changes to the existing architecture and use hardware virtualization [24] to make them a reality. The changes required in the x86 architecture to produce a Harvard architecture are fairly straight forward modifications to the paging system.

Currently, x86 implements paging by having a separate pagetable for each process and having the operating system maintain a register (CR3) that points to the pagetable for the currently running process. One pagetable is used for the process' entire address space, both code and data. In order to construct a Harvard architecture, one would need to maintain two different pagetables, one for code and one for data. As such, our proposed change to the x86 architecture to allow it to create a Harvard architecture is to create an additional pagetable register in order that one can be used for code (CR3-C) and the other for data (CR3-D). Whenever an instruction fetch occurs, the processor uses CR3-C to translate the virtual address, while for data reads and writes CR3-D is used. An operating system, therefore, would simply need to maintain two separate pagetables for each process. This capability would also offer backwards compatibility at the process level, as the operating system could simply maintain one pagetable and point both registers to it if a process requires a von Neumann architecture. We note that no changes would need to be made to the processor's translation lookaside buffer (TLB) as modern x86

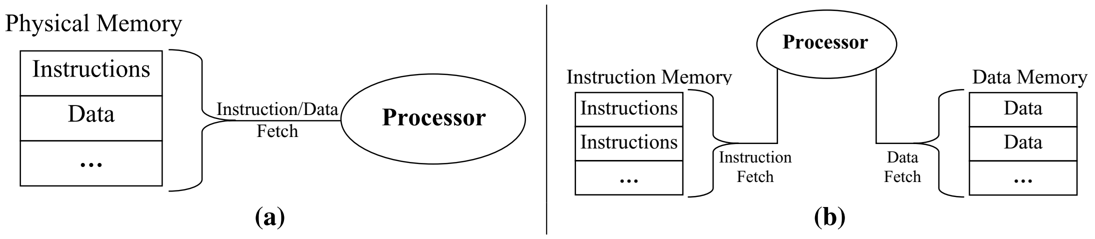

Figure 1. (a) von Neumann architecture. (b) Harvard architecture

processors already have a separate TLB for code and data.

While this approach to the problem may be effective, the requirement that the protected system be run on top of hardware virtualization inhibits its practicality. As such, another approach is needed.

Exploiting x86

Another technique for creating this Harvard architecture is to make unconventional use of some of the architecture's features in order to create the appearance of a memory that is split between code and data. Through careful use of the pagetable and the TLBs on x86, it is possible to construct a Harvard memory architecture at the process level using only operating system level modifications. No modifications need to be made to the underlying x86 architecture, and the system can be run on conventional x86 hardware without the need for hardware virtualization as in the previous method.

In the following sections we will further describe this technique as well as its unique advantages.

4. Split Memory: A Harvard Architecture on x86

Now that we have established that it is our intention to exploit, not change, the x86 architecture in order to create a virtual split memory system, we will now describe the technique in greater detail.

4.1. Virtualizing Split Memory on x86

In order to speed up pagetable lookup time, many processors include a small hardware cache called a translation lookaside buffer (TLB) which is used to cache pagetable entries. In order to better exploit locality, modern processors actually split the TLB into *two TLBs*, one for code and one for data. This feature can be exploited by a keen operating system to route data accesses for a given virtual address to one physical page, while routing instruction fetches to another. By desynchronizing the TLBs and having each contain a different mapping for the same virtual page, every virtual page may have two corresponding physical pages: One for code fetch and one for data access. In essence, a system is produced where any given virtual memory address could be routed to two possible physical memory locations. This creates a split memory architecture, as illustrated in Figure 2.

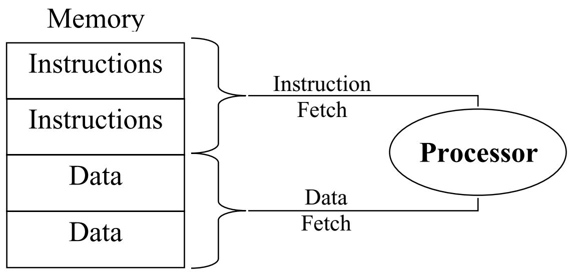

Figure 2. Split memory architecture

This split memory architecture is an environment wherein an attacker can exploit a vulnerable program and inject code into its memory space, *but never be able to actually fetch it for execution*. This is because the physical page that contains the data the attacker managed to write into the program is not accessible during an instruction fetch, as instruction fetches will be routed to an un-compromised code page. This also creates the unique opportunity to support and protect pages that contain both code and data by keeping the two physically separated but logically combined.

What to Split

Before we discuss the technical details behind successfully splitting a given page, it is important to note that different pages in a process' address space may be chosen to split based on how our system will be used.

One potential use of the system is to augment the existing non-executable page methods by expanding their protection to allow for protecting mixed code and data pages. Under this usage of the system, the majority of pages under a process' address space would be protected using the non-executable pages, while the mixed code and data pages would be protected using our technique. Using this scheme, chances are high that only a few of the process' pages would need to be protected using our method. Note that this assumes we have a good understanding of the memory space of the program being protected.

Another potential use of our system, and the one which we use in our prototype in section 5.1, is to protect every page in a process' memory space. This is a more comprehensive type of protection than simply augmenting existing schemes. Note that in this case, more pages are chosen to

Algorithm 1: Split memory page fault handler

Input: Faulting Address (addr), CPU instruction pointer (EIP), Pagetable Entry for addr (pte)

```
1  if addr == EIP then              /* Code Access */
2      pte = the_code_page;
3      unrestrict(pte);
4      enable_single_step();
5      return;
6  else                             /* Data Access */
7      pte = the_data_page;
8      unrestrict(pte);
9      read_byte(addr);
10     restrict(pte);
11     return;
12 end
```

Algorithm 2: Debug interrupt handler

Input: Pagetable Entry for previously faulting address (pte)

```
1 if processor is in single step mode then
2     restrict(pte);
3     disable_single_step();
4 end
```

be split and thus protected.

How to Split

Once it is determined which pages will be split, the technique for splitting a given page is as follows:

1) On program start-up, the page that needs to be split is duplicated. This produces two copies of the page in physical memory. We choose one page to be the target of instruction fetches, and the other to be the target of data accesses.
2) The pagetable entry (PTE) corresponding to the page we are splitting is set to ensure a page fault will occur on a TLB miss. In this case, the page is considered *restricted*, meaning it is only accessible when the processor is in supervisor mode. We accomplish it by setting or enabling the *supervisor* bit [3] in the PTE for that page. If *supervisor* is marked in a PTE and a user-level process attempts to access that page for any reason, a page fault will be generated and the corresponding page fault handler will be automatically invoked.
3) Depending on the reasons for the page fault, i.e., either this page fault is caused by a data TLB miss or it is caused by an instruction TLB miss, the page fault handler behaves differently. Note that for an instruction-TLB miss, the faulting address (saved in the *CR2* register [3]) is equal to the program counter (contained in the EIP register); while for a data-TLB miss, the page fault address is different from the program counter. In the following, we describe how different TLB misses are handled. The algorithm is outlined in algorithm 1.

Loading the Data-TLB

The data-TLB is loaded using a technique called a pagetable walk, which is a procedure for loading the TLB from within the page fault handler. The pagetable entry (PTE) in question is set to point to the data page for that address, the entry is unrestricted (we unset the *supervisor* bit in the PTE), and a read off of that page is performed. As soon as the read occurs, the memory management unit in the hardware reads the newly modified PTE, loads it into the data-TLB, and returns the content. At this point the data-TLB contains the entry to the data page for that particular address while the instruction-TLB remains untouched. Finally, the PTE is restricted again to prevent a later instruction access from improperly filling the instruction-TLB. Note that even though the PTE is restricted, later data accesses to that page can occur unhindered because the data-TLB contains a valid mapping. This loading method is also used in the PAX [2] protection model and is known to bring the overhead for a data-TLB load down to reasonable levels.

In algorithm 1 this process can be seen in lines 7–11. First, the pagetable entry is set to point to the data page and unrestricted by setting the entry to be user accessible instead of supervisor accessible. Next, a byte on the page is touched, causing the hardware to load the data-TLB with a pagetable entry corresponding to the data page. Finally, the pagetable entry is re-protected by setting it into supervisor mode once again.

Loading the Instruction-TLB

The loading of the instruction-TLB has additional complications compared to that of the data-TLB, namely because there does not appear to be a simple procedure such as a pagetable walk that can accomplish the same task. Despite these complications, however, a technique introduced in [10] can be used to load the instruction-TLB on the x86.

Once it is determined that the instruction-TLB needs to be loaded, the PTE is unrestricted, the processor is placed into single step mode, and the faulting instruction is restarted. When the instruction runs this time the PTE is read out of the pagetable and stored in the instruction-TLB. After the instruction finishes then the single step mode of the processor generates an interrupt, which is used as an opportunity to restrict the PTE.

This functionality can be seen in algorithm 1 lines 2–5 as well as in algorithm 2. First, the PTE is set to point to the corresponding code page and is unprotected. Next, the processor is placed into single step mode and the page fault handler returns, resulting in the faulting instruction being restarted. Once the single step interrupt occurs, algorithm

2 is run, effectively restricting the PTE and disabling single step mode.

4.2. Effects on Code Injection

A split memory architecture produces an address space where data cannot be fetched by the processor for execution. For an attacker attempting a code injection, this will prevent him from fetching and executing any injected code. A sample code injection attack attempt on a split memory architecture can be seen in Figure 3 and described as follows:

1. The attacker injects his code into a string buffer starting at address `0xbf000000`. The memory writes are routed to physical pages corresponding to data.
2. At the same time as the injection, the attacker overflows the buffer and changes the return address of the function to point to `0xbf000000`, the expected location of his malicious code.
3. The function returns and control is transferred to address `0xbf000000`. The processor's instruction fetch is routed to the physical pages corresponding to instructions.
4. The attacker's malicious code is not on the instruction page (the code was injected as data and therefore routed to a different physical page) and is not run. In all likelihood, the program simply crashes.

4.3. Overhead

This technique of splitting memory does not come without a cost, there is some overhead associated with the methodologies described above.

One potential problem is the use of the processor's single step mode for the instruction-TLB load. This loading process has a fairly significant overhead due to the fact that two interrupts (the page fault and the debug interrupt) are required in order to complete it. This overhead ends up being minimal overall for many applications due to the fact that instruction-TLB loads are fairly infrequent, as it only needs to be done *once* per page of instructions.

Another problem is that of context switches in the operating system. Whenever a context switch (meaning the OS changes running processes) occurs, the TLB is flushed. This means that every time a protected process is switched out and then back in, any memory accesses it makes will trigger a page fault and subsequent TLB load. The overheard of these TLB loads is significantly higher than a traditional page fault, and hence causes the majority of our slowdown. The problem of context switches is, in fact, the greatest cause of overhead in the implemented system. The experimental details of the overhead can be seen in section 5.3.

5. Implementation and Evaluation

5.1. Proof of Concept Implementation

An x86 implementation of the above method has been created by modifying version 2.6.13 of the Linux kernel. In this section, we present a description of the modifications to create the architecture.

Modifications to the ELF Loader

ELF is a format that defines the layout of an executable file stored on disk. The ELF loader is used to load those files into memory and begin executing them. This work includes setting up all of the code, data, bss, stack, and heap pages as well as bringing in most of the dynamic libraries used by a given program.

The modifications to the loader are as follows: After the ELF loader maps the code and data pages from the ELF file, for each one of those pages two new, side-by-side, physical pages are created and the original page is copied into both of them. This effectively creates two copies of the program's memory space in physical memory. The pagetable entries corresponding to the code and data pages are changed to map to one of those copies of the memory space, leaving the other copy unused for the moment. In addition, the pagetable entries for those pages get the supervisor bit cleared, placing that page in supervisor mode in order to be sure a page fault will occur when that entry is needed. A previously unused bit in the pagetable entry is used to signify that the page is being split. In total, about 90 lines of code are added to the ELF loader.

In this particular implementation of split memory the memory usage of an application is effectively doubled, however this limitation is *not* one of the technique itself, but instead of the prototype. A system can be envisioned based on demand-paging (only allocating a code or data page when needed) instead of the current method of proactively duplicating every virtual page. We would anticipate this optimization to not have any noticeable impact on performance.

Modifications to the Page Fault Handler

Under Linux, the page fault (PF) handler is called in response to a hardware generated PF interrupt. The handler is responsible for determining what caused the fault, correcting the problem, and restarting the faulting instruction.

For our modifications to the PF handler we simply modify it to handle a new reason for a PF: There was a permissions problem caused by the supervisor bit in the PTE. We must be careful here to remember that not every PF on a split page is necessarily our fault, some PFs (such as ones involving copy-on-write), despite being on split memory pages, must be passed on to the rest of the PF handler instead of being serviced in a split memory way. If it is determined that the fault was caused by a split memory page and

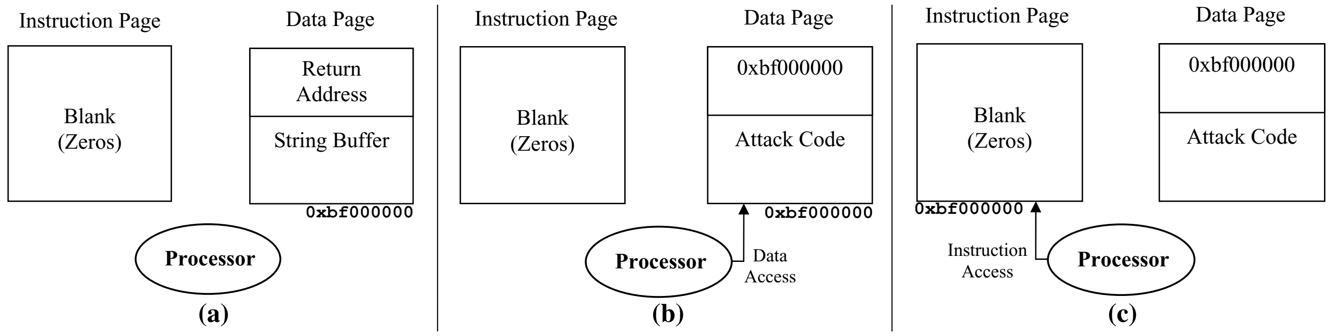

Figure 3. (a) Before the attacker injects code (b) The injection to the data page (c) The execution attempt that gets routed to the instruction page

that it does need to be serviced, then the instruction pointer is compared to the faulting address to decide whether the instruction-TLB or data-TLB needs to be loaded. (Recall from algorithm 1 that this is done by simply checking if the two are the same.)

If the data-TLB needs to be loaded, then the PTE is set to user mode, a byte on the page is touched, and the PTE is set back to supervisor mode. This pagetable walk loads the data-TLB[1]. In the event the instruction-TLB needs to be loaded, the PTE is set to user mode (to allow access to the page) and the trap flag (single-step mode) bit in the EFLAGS register is set. This will ensure that the debug interrupt handler gets called after the instruction is restarted. Before the PF handler returns and that interrupt occurs, however, a little bit of bookkeeping is done by saving the faulting address into the process' entry in the OS process table in order to pass it to the debug interrupt handler.

In total there were about 110 lines of code added to the PF handler to facilitate splitting memory.

Modifications to the Debug Interrupt Handler

The debug interrupt handler is used by the kernel to handle interrupts related to debugging. For example, using a debugger to step through a running program or watch a particular memory location makes use of this interrupt handler. For the purposes of split memory, the handler is modified to check the process table to see if a faulting address has been given, indicating that this interrupt was generated because the PF handler set the trap flag. If this is the case, then it is safe to assume that the instruction which originally caused the PF has been restarted and successfully executed (meaning the instruction-TLB has been filled) and as such the PTE is set to supervisor mode once again and the trap flag is cleared. In total, about 40 lines of code were added to the debug interrupt handler to accommodate these changes.

[1]Occasionally the pagetable walk does not successfully load the data-TLB. In this case, single stepping mode (like the instruction-TLB load) must be used.

Modifications to the Memory Management System

There are a number of features related to memory management that must be slightly modified to properly handle our system. First, on program termination any split pages must be freed specially to ensure that both physical pages (the code page and data page) get put back into the kernel's pool of free memory pages. This is accomplished by simply looking for the split memory PTE bit that was set by the ELF loader above, and if it is found then freeing two pages instead of just one.

Another feature in the memory system that needs to be updated is the copy-on-write (COW) mechanism. COW is used by Linux to make `fork`ed processes run more efficiently. That basic idea is that when a process makes a copy of itself using `fork` both processes get a copy of the original pagetable, but with every entry set read-only. Then, if either process writes to a given page, the kernel will give that process its own copy. (This reduces memory usage in the system because multiple processes can share the same physical page.) For split memory the COW system must copy *both* pages in the event of a write, instead of just one.

A update similar to the COW update is also made to the demand paging system. Demand paging basically means that a page is not allocated until it is required by a process. In this way a process can have a large amount of available memory space (such as in the BSS or heap) but only have physical pages allocated for portions it actually uses. The demand paging system was modified to allocate two pages instead of just the one page it normally does.

Overall, about 75 lines of code were added to handle these various parts related to memory management.

5.2. Effectiveness

The sample implementation was tested for its effectiveness at preventing code injection attacks using a benchmark originally put forth by Wilander et al [8]. The benchmark was modified slightly in order to allow it to handle having the code injected on the data, bss, heap, and stack portions of the program's address space. In addition, four of

Table 1. The number of attacks halted when code is injected onto the data, bss, heap, and stack segments

Attack Type	Hijack Type	Injection Destination			
		Data	BSS	Heap	Stack
Buffer overflow on stack	Return address	✓	✓	✓	✓
	Old base pointer	✓	✓	✓	✓
	Function pointer as local variable	✓	✓	✓	✓
	Function pointer as parameter	✓	✓	✓	✓
	Longjmp buffer as local variable	✓	✓	✓	✓
	Longjmp buffer as function parameter	✓	✓	✓	✓
Buffer overflow on heap/bss	Function pointer	✓	✓	✓	✓
	Longjmp buffer	✓	✓	✓	✓
Buffer overflow of pointers on stack	Return address	N/A	N/A	✓	N/A
	Old base pointer	N/A	N/A	N/A	N/A
	Function pointer as local variable	✓	✓	✓	✓
	Function pointer as parameter	✓	✓	✓	✓
	Longjmp buffer as local variable	✓	✓	✓	✓
	Longjmp buffer as function parameter	✓	✓	✓	✓
Buffer overflow on heap/bss	Return address	N/A	N/A	✓	N/A
	Old base pointer	N/A	N/A	N/A	N/A
	Function pointer as variable	✓	✓	✓	✓
	Longjmp buffer as variable	✓	✓	✓	✓

the testcases did not successfully execute an attack on our unprotected system, and so have been labeled "N/A." Table 1 shows the results of running the benchmark. The checkmarks indicate that the system successfully halted the attack. As can be seen, the system was effective in preventing all types of code injection attacks present in the benchmark. The effectiveness of the system is due to the fact that no matter what method of control-flow hijacking the benchmark uses, the processor is simply unable to fetch the injected code.

5.3. Performance

A number of benchmarks, both applications and microbenchmarks, were used to test the performance of the system. Our testing platform was a modest system, a Pentium III 600Mhz with 384 MB of RAM and a 100MBit NIC. When applicable, benchmarks were run 10 times and the results averaged. Details of the configuration for the tests are available in table 2. Each result has been normalized with respect to the speed of the unprotected system.

Four benchmarks that we consider to be a reasonable assessment of the system's performance can be found in Figure 4. First, the Apache [25] webserver was run in a threading mode to serve a 32KB page (roughly the size of Purdue University's main index.html). The ApacheBench program was then run on another machine connected via the NIC to determine the request throughput of the system as a whole. The protected system achieved a little over 89% of the unprotected system's throughput. Next, gzip was used to compress a 256 MB file, and the operation was timed. The protected system was found to run at 87% of full speed. Third,

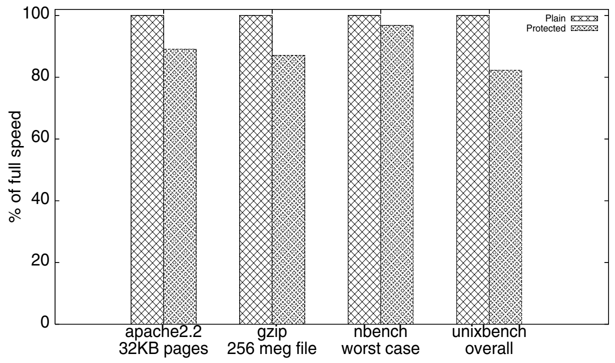

Figure 4. Normalized performance for applications and benchmarks

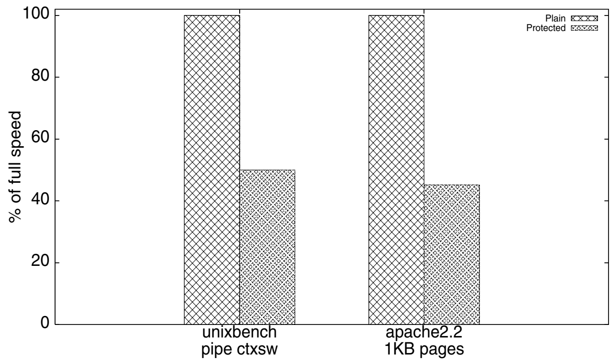

Figure 5. Stress-testing the performance penalties due to context switching

Table 2. Configuration information used for performance evaluation

Item	Version	Configuration
Slackware	10.2.0	Using Linux 2.6.13
Apache	2.2.3	Worker mpm mode, set to spawn one process with threads
ApacheBench	2.0.41-dev	`-c3 -t 60 <url/file>`
Unixbench	4.1.0	N/A
Nbench	2.2.2	N/A
Gzip	1.3.3	Compress a 256 MB file.

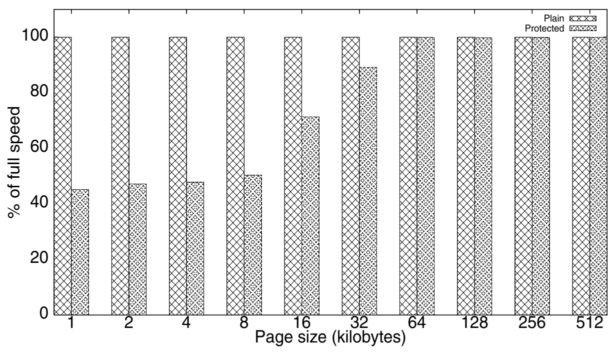

Figure 6. Closer look into Apache performance

the nbench [26] suite was used to show the performance under a set of primarily computation based tests. The slowest test in the nbench system came in at just under 97%. Finally, the Unixbench [27] unix benchmarking suite was used as a micro-benchmark to test various aspects of the system's performance at tasks such as process creation, pipe throughput, filesystem throughput, etc. Here, the split memory system ran at 82% of normal speed. This result is slightly disappointing, however it can be easily explained by looking at the specific test which performed poorly, which we do below. As can be seen from these four benchmarks, the system has very reasonable performance under a variety of tasks.

If we simply left our description of the system's performance to these four tests, some readers may object that given the description of the system so far and the mention in section 4.3 of the various sources of overhead, something must be missing from our benchmarks. As such, two benchmarks contrived to highlight the system's weakness can be found in Figure 5. First, one of the Unixbench testcases called "pipe based context switching" is shown. This primarily tests how quickly a system can context switch between two processes that are passing data between each other. The next test is Apache used to serve a 1KB page. In this configuration, Apache will context switch heavily while serving requests. In both of these tests, context switching is taken to an extreme and therefore our system's performance degrades substantially due to the constant flushing of the TLB. As can be seen in the graph, both are at or below 50%. In addition, in Figure 6, we have a more thorough set of Apache benchmarks demonstrating this same phenomena, namely that for low page sizes the system context switches heavily and performance suffers, where as for larger page sizes that cause Apache to spend more time on I/O as well as begin to saturate the system's network link, the results become significantly better. These tests show very poor performance, however we would like to note that they are shown here to be indicative of the system's worst case performance under highly stressful (rather than normal) conditions.

Overall, the system's performance is reasonable, in most cases being between 80 and 90% of an unprotected system. Moreover, if split memory was supported at the hardware level as described in section 3.3, the overheard would be almost non-existent. Based on previous work [28], we also have reason to believe that building the split memory system on top of an architecture with a software loaded TLB, such as SPARC, would also provide further performance improvements.

6. Limitations

There are a few limitations to our approach. First, when an attack is stopped by our system the process involved will crash. We offer no attempt at any sort of recovery. This means an attacker can still exploit flaws to mount denial-of-service attacks. Second, as shown in other work [29], a split memory architecture does not lend itself well to handling self-modifying code. As such, self-modifying programs cannot be protected using our technique. Next, this protection scheme offers no protection against attacks which do not rely on executing code injected by the attacker. For example, modifying a function's return address to point to a different part of the original code pages will not be stopped by this scheme. Fortunately, address space layout randomization [14] could be combined with our technique to help prevent this kind of attack. Along those same lines, non-control-data attacks [23], wherein an attacker modifies a program's data in order to alter program flow, are also not protected by this system. We have also not analyzed the system's functionality on programs that include dynamically loadable modules (such as DLL files on windows) but do not anticipate that such programs would be difficult to support.

7. Conclusions

In this paper, we present an architectural approach to prevent code injection attacks. Instead of maintaining the traditional single memory space containing both code and data, which is often exploited by code injection attacks, our approach creates a split memory that separates code and data into different memory spaces. Consequently, in a system protected by our approach, code injection attacks may result in the injection of attack code into the data space. However, the attack code in the data space can not be fetched for execution as instructions are only retrieved from the code space. We have implemented a Linux prototype on the x86 architecture, and experimental results show the system is effective in preventing a wide range of code injection attacks while incurring acceptable overhead.

8. Acknowledgments

We would like to thank Glenn Wurster as well as the anonymous reviewers for their helpful comments and suggestions. This work was supported in part by NSF Grants OCI-0438246, OCI-0504261, and CNS-0546173.

References

[1] A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003. *http://support.microsoft.com/kb/875352*. Last accessed Dec 2006.

[2] Pax pageexec documentation. *http://pax.grsecurity.net/docs/pageexec.txt*. Last accessed Dec 2006.

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

[4] Buffer overflow attacks bypassing dep (nx/xd bits) - part 2 : Code injection. *http://www.mastropaolo.com/?p=13*. Last accessed Dec 2006.

[5] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[6] H. Etoh. Gcc extension for protecting applications from stack-smashing attacks. *http://www.trl.ibm.com/projects/security/ssp/*. Last accessed Dec 2006.

[7] Vendicator. Stack shield: A "stack smashing" technique protection tool for linux. *http://www.angelfire.com/sk/stackshield/info.html*. Last accessed Dec 2006.

[8] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, San Diego, California, February 2003.

[9] J. von Neumann. First draft of a report on the edvac. 1945. Reprinted in *The Origins of Digital Computers Selected Papers*, Second Edition, pages 355–364, 1975.

[10] P. C. van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Trans. Dependable Secur. Comput.*, 2(2):82–92, 2005.

[11] H. H. Aiken. Proposed automatic calculating machine. 1937. Reprinted in *The Origins of Digital Computers Selected Papers*, Second Edition, pages 191–198, 1975.

[12] H. H. Aiken and G. M. Hopper. The automatic sequence controlled calculator. 1946. Reprinted in *The Origins of Digital Computers Selected Papers*, Second Edition, pages 199–218, 1975.

[13] kernelthread.com: Securing memory. *http://www.kernelthread.com/publications/security/smemory.html*. Last accessed Dec 2006.

[14] Pax aslr documentation. *http://pax.grsecurity.net/docs/aslr.txt*. Last accessed Dec 2006.

[15] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. *12th USENIX Security*, 2003.

[16] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. *14th USENIX Security*, 2005.

[17] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent Runtime Randomization for Security. *In Proc. of 22nd Symposium on Reliable and Distributed Systems (SRDS) , Florence, Italy*, Oct. 2003.

[18] 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. *10th ACM CCS*, 2003.

[19] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. *10th ACM CCS*, 2003.

[20] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. *USENIX Annual Technical Conference*, 2005.

[21] L. Lam and T. Chiueh. Checking Array Bound Violation Using Segmentation Hardware. *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 388–397, 2005.

[22] Wind river: Vxworks. *http://www.windriver.com/vxworks/*. Last accessed Mar 2007.

[23] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. In *Proc. USENIX Security Symposium*, aug 2005.

[24] bochs: The open source ia-32 emulation project. *http://bochs.sourceforge.net/*. Last accessed Dec 2006.

[25] The apache http server project. *http://httpd.apache.org/*. Last accessed Dec 2006.

[26] Linux/unix nbench. *http://www.tux.org/~mayer/linux/bmark.html*. Last accessed Dec 2006.

[27] Unixbench. *http://www.tux.org/pub/tux/benchmarks/System/unixbench/*. Last accessed Dec 2006.

[28] G. Wurster. A generic attack on hashing-based software tamper resistance. Master's thesis, Carleton University, Canada, Apr 2005.

[29] J. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*, pages 18–27, Tucson, AZ, USA, Dec. 2005. Applied Computer Associates, IEEE.