8-2008

Extending Hardware Based Mandatory Access Controls to Multicore Architectures

Brian Lewis Sharp
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Brian Lewis Sharp entitled "Extending Hardware Based Mandatory Access Controls to Multicore Architectures." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Dr. Greg Peterson, Major Professor

We have read this thesis and recommend its acceptance:

Dr. David Icove, Dr. David Straight

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Brian Lewis Sharp entitled "Extending Hardware Based Mandatory Access Controls to Multicore Architectures." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

_____
Dr. Greg Peterson, Major Professor

We have read this thesis

and recommend its acceptance:

Dr. David Icove_____

Dr. David Straight_____

Accepted for the Council:

Carolyn R. Hodges_____

Vice Provost and Dean of the
Graduate School

(Original Signatures are on file with official student records.)

Extending Hardware Based Mandatory Access Controls to Multicore Architectures

A Thesis Presented for
the Master of Science
Degree
The University of Tennessee, Knoxville

Brian Lewis Sharp
August 2008

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my parents for showing me the importance of education at an early age. They have sacrificed more than I could ever know to give me the opportunity to succeed. I will be forever grateful for their support in my every undertaking. The only way I could ever hope to repay them would be to give the same gift to my children.

I also owe a great deal of gratitude to Dr. Peterson who has put up with me for the past four years. Over the years he has played the role of academic advisor, professor, summer internship mentor (twice), facilitator of another summer internship, major professor, mentor, and friend. Without him constantly pushing me to take more and difficult classes, I would never have finished my BS in four years (with six hours of graduate credit), and my MS in one year. By setting me up with a summer internship at the Air Force Research Lab he sparked my interest in computer security. His advice on academic, career, and personal matters has been and continues to be priceless.

I would also like to thank members of my committee, Dr. Icove and Dr. Straight. Taking on another Masters student on such short notice indicates their sincere interest in helping out the students of this great university. Dr. Icove's tips on passing the Fundamentals of Engineering Exam proved to be quite useful come test time. Dr. Straight's unique approach to Systems Programming helped me to tackle problems in a different manner and motivated me to attend class.

If it were not for the fine folks at the Air Force Research Lab in Rome, NY, I would still be in school right now. This work was funded by award number

FA8750-06-1-0185. I would like to thank Lok Kwong Yan for answering all my questions about everything from computer security to the pros and cons of government employment. As my mentor in the summer of 2007, he actively participated in my project including contributing code to my work (which is unheard from most mentors). He was also nice enough to put up with my crippling addiction to Mountain Dew. I'd like to thank Steve Drager for offering me the chance to continue my work which was started there. If it were not for this funding, I would have been a teaching assistant for freshman engineering classes. In addition to that grim fate, it would have also pushed my graduation date back at least one semester, and probably a full year.

All along my journey I have been helped by friends, family, and teachers. If I were to name each person and their contribution to my success, this thesis would be well over the maximum length allowed. To them I must simply say, thank you, and that I will continue to make the best of every opportunity I am given.

ABSTRACT

Memory based vulnerabilities have plagued the computer industry since the release of the Morris worm twenty years ago. In addition to buffer overflow attacks like the Morris worm, format strings, ret-libC, and heap double free() viruses have been able to take advantage of pervasive programming errors. A recent example is the unspecified buffer overflow vulnerability present in Mozilla Firefox 3.0. From the past one can learn that these coding mistakes are not waning. A solution is needed that can close off these security shortcomings while still being of minimal impact to the user. Antivirus software makers continuously overestimate the lengths that the everyday user is willing to go to in order to protect his or her system. The ideal protection scheme will be of little or no inconvenience to the user. A technique that fits this niche is one that is built into the hardware. Typical users will never know of the added protection they're receiving because they are getting it by default. Unlike the NX bit technology in modern x86 machines, the correct solution should be mandatory and uncircumventable by user programs. The idea of marking memory as non-executable is maintained but in this case the granularity is refined to the byte level. The standard memory model is extended by one bit per byte to indicate whether the data stored there is trusted or not. While this design is not unique in the architecture field, the issues that arise from multiple processing units in a single system causes complications. Therefore, the purpose of this work is to investigate hardware based mandatory access control mechanisms that work in the multicore paradigm. As a proof of concept, a buffer overflow style attack has

been crafted that results in an escalation of privileges for a nonroot user. While effective against a standard processor, a CPU modified to include byte level tainting successfully repels the attack with minimal performance overhead.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

1. Introduction

The world of computer security is an ongoing cat and mouse game between antivirus companies and hackers. In this game, the black hats seem to have the upper hand as most patches for malicious software are not available until after the attack is unleashed on an unfortunate group of computers. While updates are necessary to fix existing problems in already released software, a more proactive approach to computer security could greatly benefit the everyday user. This could come in the form of a variety of solutions. One of the more promising and interesting areas is mandatory access controls. Mandatory access controls (MACs) in computer systems are memory access protocols that the user cannot circumvent even if he or she wanted. This can be contrasted with discretionary controls like the standard read, write, and execute properties that are common to Windows and Linux platforms. What makes correctly used MACs powerful is the fact that a user, whether due to ignorance or ill intent, cannot cause harm to the system. This ability to repel an attack from an insider has led the National Security Agency to create and adopt a MAC based operating system known as SELinux [35]. Hardware based MACs can first be seen in an elaborate new computer architecture introduced in 1960 [24]. Failing to reach the mainstream, they have been the subject of academic research ever since. These systems have the ability to enforce security protocols at the lowest level and with little to no change to existing software which makes the solution both powerful and far-reaching. While meeting security goals is important, a successful computer security solution must also conform to the paradigm of the future.

The difficulty in continuously increasing clock speed has caused CPU designers to resort to increasing the number of cores on a chip in an effort to maintain exponential growth in computing power. The popularity of this trend can be seen in a quote from Intel President Paul Otellni, “We are dedicating all our future product development to multicore designs” [16]. The result of this has been a programming and security nightmare. The challenge of merely maintaining consistency across memory has proved to be difficult. A security solution that doesn’t fit with a multicore design is inherently doomed. Therefore, it is the purpose of this work to adapt hardware based mandatory access controls to fit multicore architectures.

1.1. Motivation

1.1.1. Buffer Overflow Attack Prevalence

Of all the different ways for a hacker to gain access to privileged information or to disrupt desired functionality, buffer errors are probably the easiest and most successful. Buffer overflows are just one form of memory based vulnerability, however, according to the National Vulnerability Database (NVD) they still account for around 12% of all computer vulnerabilities [9] . The number of vulnerabilities classified as “buffer errors” by the NVD is given in Figure 1.1. It is important to note in Figure 1.1 that data has only been collected up to the midpoint of 2008. Extrapolating the year’s trend indicates that in 2008 there will be over 600 buffer error vulnerabilities. The percentage of all vulnerabilities which fall into that same category is given in Figure 1.2.

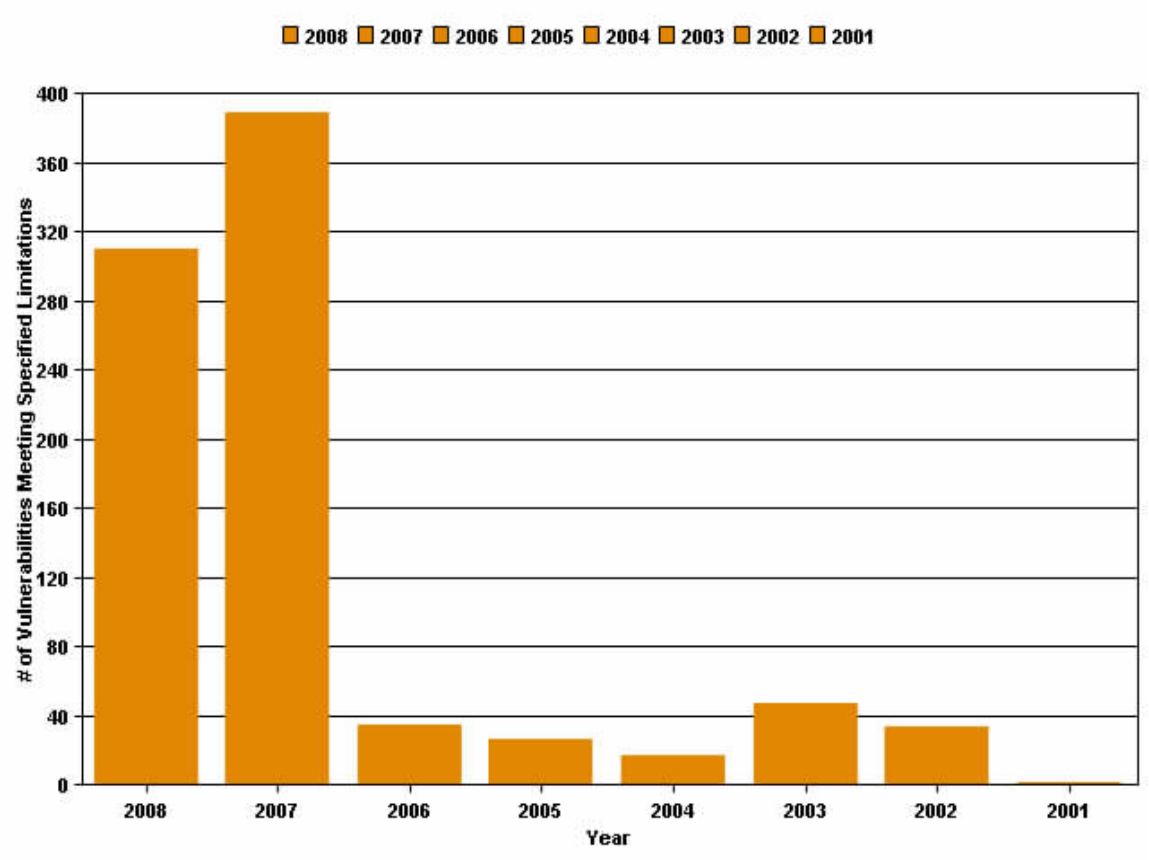

Figure 1.1: Number of vulnerabilities classified as buffer errors over time [9]

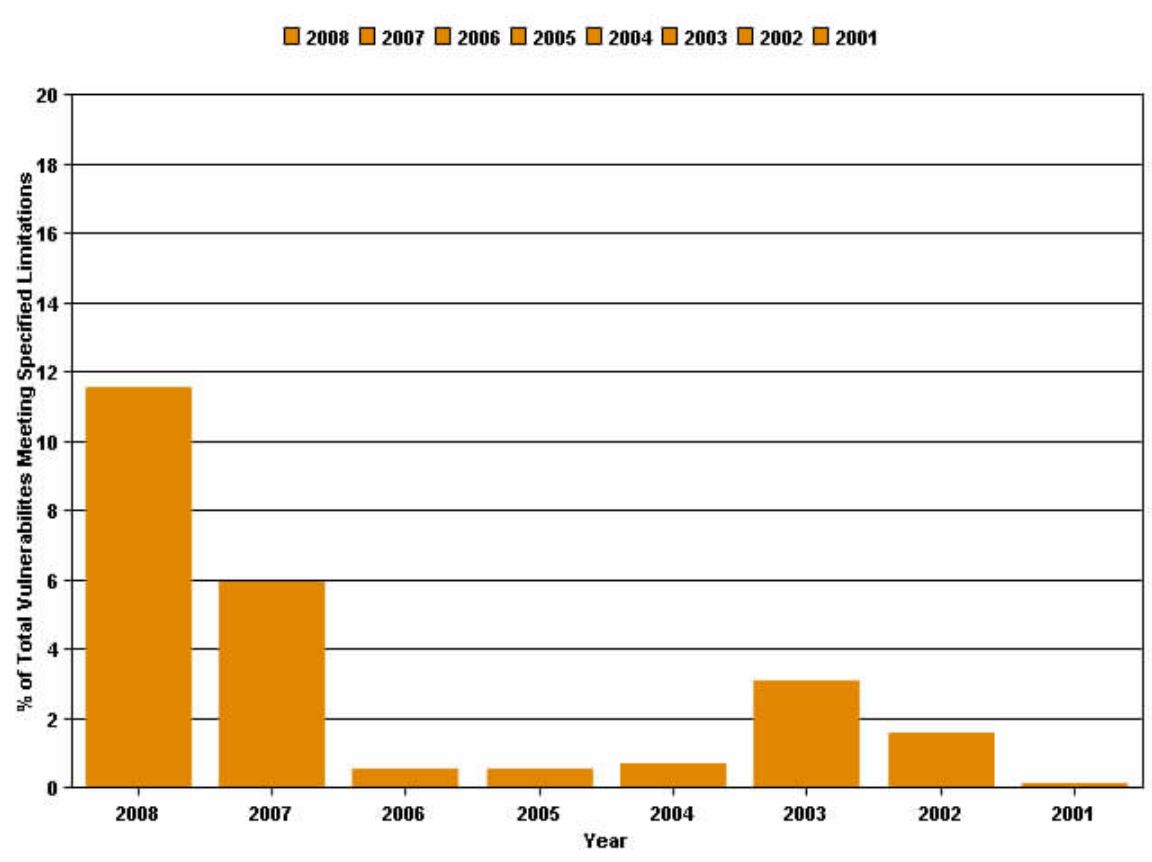

Figure 1.2: Percentage of vulnerabilities classified as buffer errors over time [9]

Even more disturbing is the fact that despite the antiquity of this problem, the percentage of all vulnerabilities that are of this type has been growing over the past couple of years. In the face of two decades of research and numerous solutions, the problem still persists. This thesis proposes a solution to the buffer overflow problem along with other, less prevalent, memory based vulnerabilities such as format strings and heap-based attacks.

1.1.2. End User Involvement

The problem is not the lack of solutions, but of a pervasive one that can be enabled and adapted easily by large populations and that actually solves the underlying problem. It can be seen from past work that the solutions with the greatest chance of adoption, and therefore success, are those that require a minimal amount of effort and skill from the user. Many types of existing solutions already require too much from their users. People may not be willing to install update software because they don't like having to restart their computer everyday due to new updates. Most users won't pay a monthly subscription fee to get the latest antivirus updates. Observations like this can be seen in a more quantitative way in Figure 1.3 [37]. Despite a plethora of new computer architectures, the leader continues to be x86 due, in part, to its large installed base of software. The millions of lines of legacy code still in use means that requiring developers to substantially change their applications or libraries often leads to difficult new architectures being labeled unusable.

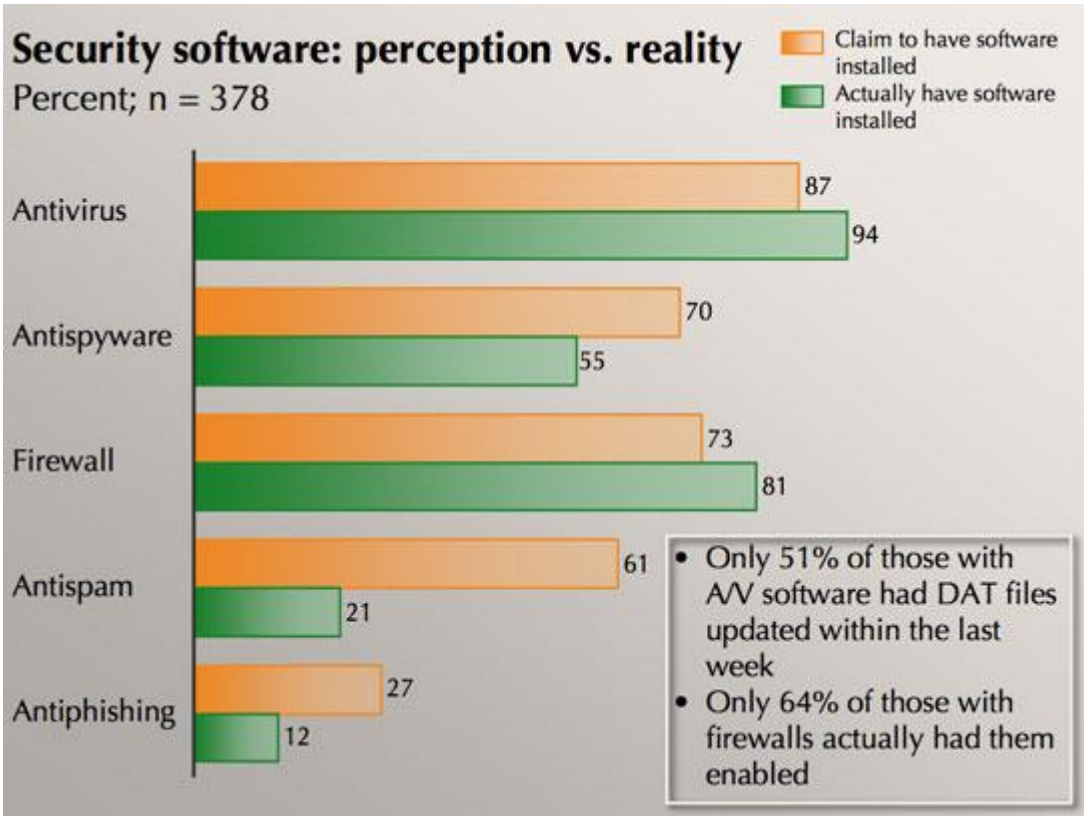

Figure 1.3: Disparity Between Installed User Security Software and Perceived [37]

Therefore, the desired solution should be completely backwards compatible with existing x86 code and require no recompilation or relinking. Often, this type of solution is one that is hardware based.

1.2. Background

The first great computer infection, the Morris Worm of 1988, made use of a buffer overflow exploit in the `fingerd` application as one of its methods of propagation [30]. Given the vastness and damage of the worm, one would think that in a short amount of time a solution would be developed that would inoculate the world's computers from such a sickness. However, in 2001, thirteen years after the Morris Worm, the Code-Red and CodeRedII worms would use another buffer overrun that would cost $2.6 billion [26]. This is a testimony to the ineffectiveness of the proposed protection schemes during that thirteen year period and the overall prevalence of this attack scheme. Unlike standard file permissions, a proper solution should not be discretionary upon the user's desire, but should have at least some mechanisms that are mandatory and uncontrollable even at the privileged operating system level. This level of security would be enforced by the processor itself, as a fundamental property of the ISA.

1.2.1. Stack Smashing Attack

One looking for a walkthrough about how to create a buffer overflow attack need look no further than the archives of the hacker magazine *Phrack* [2].

In 1998, it published a widely cited article giving a walkthrough on how to construct a buffer overflow attack that is readable at the introductory programming level. To understand such an attack, first one must consider what a standard stack frame looks like in memory. This is given in Figure 1.4. A typical stack smashing attack will attempt to find an unsafe function call like strcpy(). strcpy() copies the string pointed to by *source* into the array pointed to by *destination* up to and including the terminating null character. The function prototype for strcpy() is given in Figure 1.5. The security issue arises from the fact that strcpy() copies data into *destination* until the null terminating character is found in *source* despite the length of either arrays. Good programming practice dictates that code should be written to make sure that no more characters are copied into *destination* then it can hold. Unfortunately, good programming practices are not always observed. When a *source* array is not properly bounds checked, it allows an attack to overflow the *destination* buffer. Most often, this buffer is overflowed up to the point of overwriting the return address stored on the stack. The attacker changes the return address to point back into the buffer thereby redirecting execution into the attacker supplied string. The resulting stack frame is shown side by side with a normal one in Figure 1.6. Normally the attacker stores shell code to exec a shell in the buffer and by this means has his own prompt to do whatever he/she wishes.

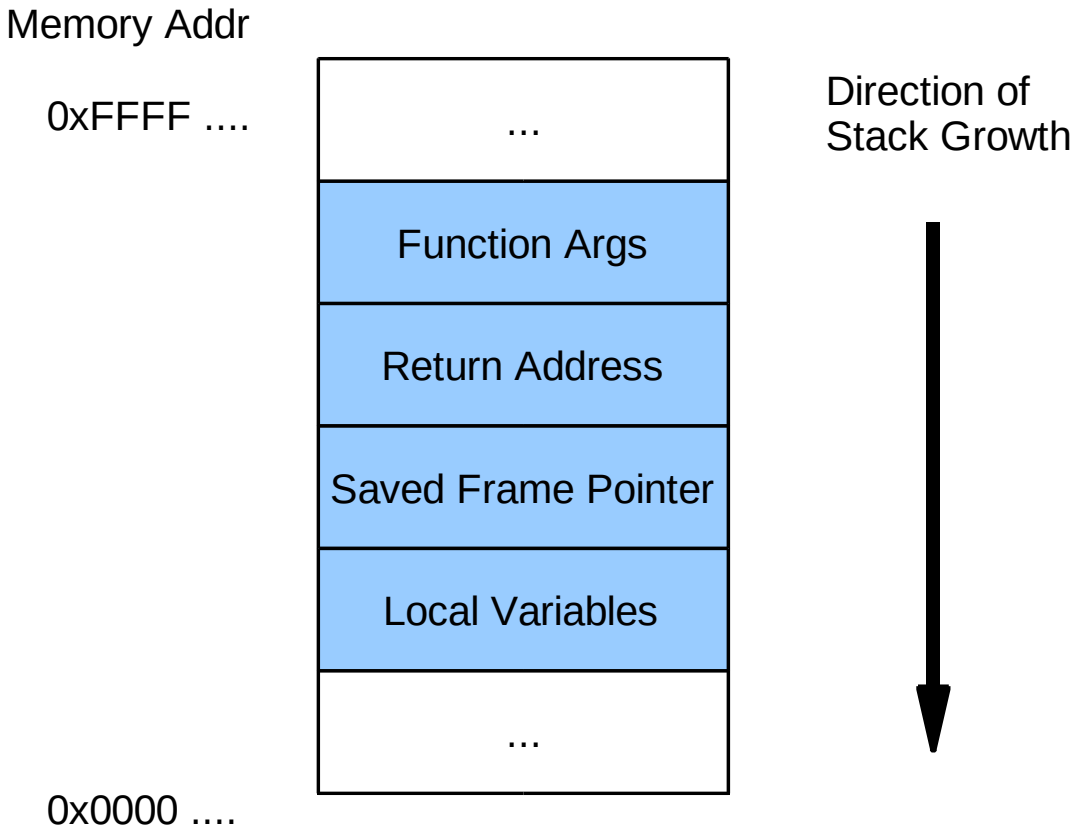

Figure 1.4: Standard Stack Frame

```
char * strcpy ( char * destination, const char *
source )
{
      ...
}
```

Figure 1.5: Function prototype for strcpy()

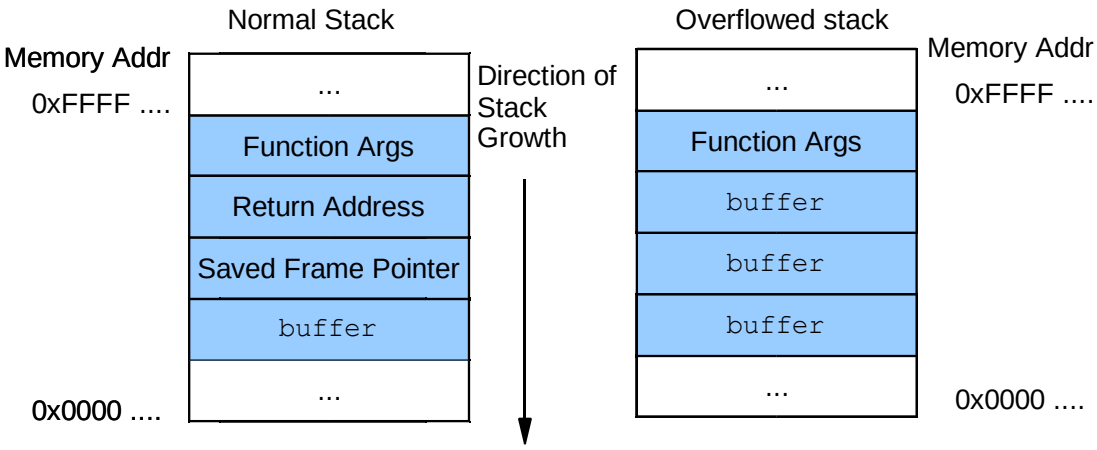

Figure 1.6: Normal and Overflowed Stack Frames

1.3. Summary

In this chapter, the need for a protection scheme was motivated by the substantial and increasing number of buffer overflow attacks. Statistics show that most users are not aware of the security features on their computers. Regardless if this is because of a lack of knowledge or lack of effort, a successful solution to memory based vulnerabilities will be one that requires little from the everyday user. A high level look into stack frames and stack buffer overflows was also offered. In an effort to thwart stack buffer overflows, many solutions have been attempted over the years, some of which will be discussed in the next chapter.

2. Related work

Since the inception of the modern computer as a device to calculate artillery trajectories, security has been a concern. With the dawn of the Internet it has taken on a whole new meaning and importance. In this chapter some historical attempts to provide security are discussed followed by modern methods that attempt to achieve a robust system.

2.1. A Brief History of Secure Architectures

Dating back to 1959, there has been an effort to create more secure processors by enforcing safety precautions at the hardware level. Capability-based architectures refer to computer systems that access data using an address that refers to both the memory object itself and a set of access rights that govern how that data can be used. For an excellent reference of such architectures see [23].

2.1.1. Burroughs

First shown in the early 1960s, the Burroughs family of processors incorporated some very sophisticated features for their time [24]. Originally, the B5000 used a 1 bit tag as part of its 32 bit word. The B6000 expanded it to 3 bits and moved it outside the word. It differentiates data from code and control words and is even used to indicate type (such as single and double precision floating point). The hardware enforced security mechanism makes it impossible to execute data as code or to interpret code as data.

2.1.2. System/38

Released in 1980, the IBM System/38 sought to be a totally object-oriented architecture [18]. The System/38 featured 40 bit words consisting of 32 data bits, 7 bit ECC, and a 1 bit tag. The tag bit is set whenever the data bits contain a pointer while all other words in memory have their tag bits cleared. These tag bits cannot be accessed by the instruction interface and cannot be set by the user. Instead, they are manipulated by instructions that use microcode to build the pointers and maintain the integrity of the tag bits. User modification of the pointer results in its tag bits being cleared thus making it invalid for addressing purposes.

2.1.3. iAPX 432

Introduced in the year after the IBM System/38, the design and layout of the chipset for the Intel iAPX 432 took over 100 man-years [23]. Memory references are done using 32 bit long access descriptors (ADs) that specify the actual address and access rights to an object. A procedure can only address and manipulate the ADs that are within its execution environment. The access rights specify whether the possessor of the AD can read from or write to the object or delete the AD itself. Unfortunately, the iAPX 432 was doomed by performance problems and an overzealous marketing campaign.

2.1.4. Unisys

A novel computer architecture that is still around today is used by Unisys Mainframes [41]. The ISA tags each word of memory to indicate how the data

stored there can be used. All data references are done through descriptors generated by the hardware and operating system using instructions unavailable to ordinary user code. Every memory reference is checked for a valid descriptor and that the reference is within appropriate bounds. Programs that are running are not given privilege to descriptors that hold their own code or that of another program. Furthermore, code and data are kept separate eliminating any adjacency between buffers and areas containing executable code.

2.1.5. NX bit

Mandatory Access Controls were brought into the mainstream when AMD began to use an extra bit, the No eXecute bit (NX), to mark pages of memory as non-executable [43]. Intel later followed with what it called the Execute Disable bit [13]. The capability of the processor to take advantage of this sort of functionality can be queried by the operating system that is running. When activated by setting the bit IA32_EFER.NXE, memory pages can be marked as not being executable. This is done by adjusting bit 63 in the corresponding page table entry for that page of memory. If the protection is running and an instruction fetch to a linear address translates to a physical address in a memory page that has the execute disable bit set, a page fault exception will be generated. This sort of protection is very close to what is desired in protecting memory from memory based vulnerabilities: there is no effort required of the user other than having a processor with the ability, it incurs very little memory or performance overhead, and it is backwards compatible with existing code.

To allow for backwards compatibility, Intel decided to give the host OS the ability to turn non-executable pages on or off. Windows XP Service Pack 2 and Windows 2003 Service Pack 1 contain patches to take advantage of this hardware feature by using what Microsoft calls Data Execution Prevention (DEP) [1]. Shortly after its debut, exploits began to be posted that easily sidestepped the mechanism [28]. In order to bypass DEP, a ret-libC style attack can be used to jump to a section of system code marked as executable that can then further be exploited to disable DEP and return into shell code stored in the original buffer. If there was no way that a process could disable NX support at runtime, this exploit would not work.

2.2. Recent Solutions

In an attempt to close off the buffer overflow attack vector, many solutions have come about. As the goal of this work is to place a minimal amount of burden on the user, previous attempts can best be organized along a continuum with the least user involvement at one end and the most at the other. Within that continuum, most solutions fall into a particular category depending upon their level of abstraction: language, compiler, library, application, operating system, or hardware. These form a gradient as shown in Figure 2.1.

2.2.1. Safe Languages

Buffer overflows arise from the lack of type safety and bounds checking in C.

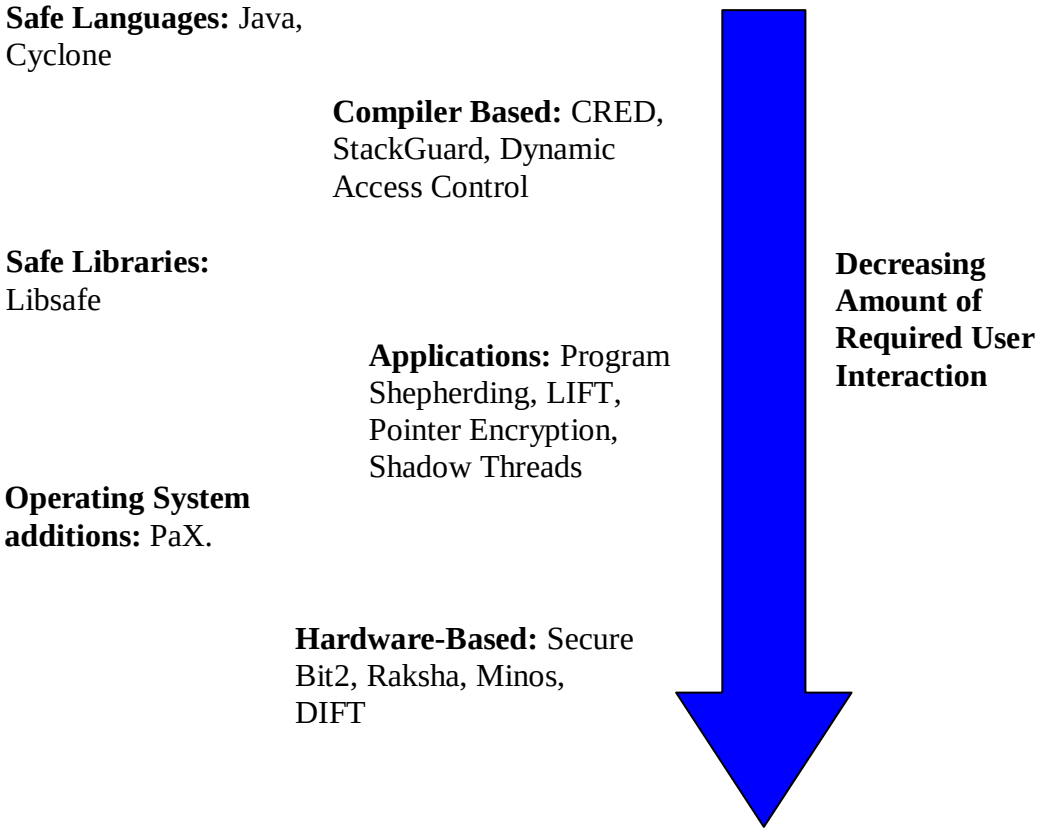

Figure 2.1: Amount of user involvement required for various type solutions.

Work has been done in the past to create a programming language that has built in bounds checking or is otherwise immune to buffer attacks. Despite whatever security features these languages may have, they normally require code to be ported from C to the new safe language. Porting this amount of legacy code, some of which has been lost, to an entirely new language would be a massive undertaking. Another problem with both language and compiler techniques is the availability of source code. While open source movements are gaining popularity, there is still a great amount of proprietary software in use. For companies unwilling to release the source code, the following techniques would not work.

2.2.1.1. Java

The security and portability features of Java make it very desirable to replace C as the language of choice. There seems to be a growing trend of applications being written in Java versus other languages [39]. The Java Virtual Machine (JVM) and corresponding bytecode is what makes Java so portable and safe [8]. Interestingly enough, the Java Virtual Machine (JVM) is written in C, and is vulnerable to buffer overflow attacks itself [12]. Unfortunately, Java as a vulnerability solution suffers from the previously stated issues related to legacy code porting.

2.2.1.2. Cyclone

Cyclone follows nearly all of C's lexical conventions and grammar. It also offers

the same fundamental and composite data types. Where they differ is on the issue of security [19]. A cyclone compiler performs a static analysis on the program and inserts runtime checks into the compiled code at places where safety is in danger. The compiler can also choose to reject a program based on a static analysis of its code that deems it unsafe. As compared to the same program written in C, Cyclone's slowdown varies from 1X to 3X and as much as 6X in pathological cases that feature a great amount of pointer arithmetic. The test suite consisted of common web utilities such as `http_get` and `http_post`, and computationally intensive benchmarks like `cfrac`, `maxtmult`, and `tile` [19]. Subsequently, Cyclone found array bounds violations in three benchmarks, one of which dates back to the mid 1980s.

2.2.2. Compiler Based Solutions

While compiler based solutions share the requirement of source code availability, they also have the unique disadvantage of demanding users to recompile their programs. Many users are unaware of how to do this and many of those who are may choose not to deal with the hassle of recompiling their programs.

2.2.2.1. StackGuard

StackGuard, part of the Immunix Linux Distribution, is probably one of the most well known stack smashing protection schemes available [6]. It is a modified version of the gcc compiler that automatically inserts "canaries" into the stack before a function is called [7]. After the function completes, and before it returns to the return address on the stack, the canary value is checked with a stored

version. If the values do not match, then it means that the canary has been illegally overwritten, and the stack corrupted. A canary may be a simple XOR with a secret key or an even simpler scheme. A canary that is comprised solely of null values is adequate to stop buffer overflow attacks. When strcpy() is called, it will return upon reaching the null value instead of continuing to overwrite out of bounds data on the stack.

2.2.2.2. CRED

CRED (C Range Error Detection) is an extension to the GNU compiler that was developed at Stanford University [33]. It relies on replacing every out-of-bounds pointer value with the address of a special OOB (out-of-bounds) object created for that value. At the current state of development, CRED would break when using an out of bounds pointer in an external library. It was effective against 20 different buffer overflow attacks and also ran 20 open-source programs consisting of 1.2 million lines of code [33]. The average performance degradation was 2X but up to 20X for some applications. Incorporating this idea into the next distribution of an operating system would allow CRED to reach a large user base without forcing users to recompile existing applications.

2.2.2.3. Dynamic Access Control

Dynamic Access Control monitors program data that might be indicative of an attack, even those attacks that do not alter control flow [44]. The dangers of these types of attacks are demonstrated in [5]. Dynamic Access Control requires support at both the hardware and micro-architecture level. The compiler

identifies program regions in which the data should not be modified as per program semantics. If there is an attempt to modify this data at runtime, the hardware detects the attack.

2.2.3. Safe Libraries

Libsafe is an example of a secure library [15]. It intercepts certain unsafe calls such as strcpy() or fgets() and calculates the maximum allowed size of the buffer based on the stack frame address, and then calls the safer bounded variant such as strncpy(). Although this solution has proven to be effective against some stack overflows, it still suffers some flaws. One specific problem is that it does not prevent the overflow itself, in-band data within the stack frame can still be overwritten. As previously mentioned, a system in which just the control flow data is protected can still be vulnerable to attack. Like other library-based approaches, it is only applicable when the application is dynamically linked. It will not be effective for statically linked software or for user defined functions.

2.2.4. Monitoring Applications

2.2.4.1. Program Shepherding

Program Shepherding is a system built on top of Runtime Introspection and Optimization (RIO), a dynamic optimizer application, and seeks to stop malicious code executions by using the concepts of restricted code origins, restricted control transfers, and uncircumventable sandboxing [20] . When an application is run under this paradigm, the loader must first determine if the block of

instructions is trusted in accordance with a security policy and the origins of the code, e.g. executable file from the disk or dynamically generated code, and tags them as executable. This first step is known as restricted code origins. Restricted control transfers refers to the restriction of jumps and branches from one block of memory to another only if it is allowed in the security policy, e.g. if the target of the branch is not tagged as executable, then control transfer should be restricted. Finally, uncircumventable sandboxing is used to ensure that all implemented security checks must be done at all times.

2.2.4.2. LIFT

Low-Overhead Information Flow Tracking is a tainting architecture that is built on top of the dynamic binary translator StarDBT [3]. A typical tainting scheme would identify data from untrusted channels (e.g. network, keyboard, USB) and taint, or tag, that data as being not executable. If this data is copied to a new location or used as an operand in another instruction, the destination operand would also be marked as untrusted. Only untainted memory locations would be used as return address, jump destinations or function pointers. For LIFT, a one bit tag is associated with each byte of data in memory or in registers. These tags are stored in a special memory region that generates a protection fault when a program attempts to access them. Tags for registers are stored in a spare CPU register. If the architecture has no spare registers, another special memory region is allocated to store the data. The extra functionality of propagating taint bits is accomplished by instrumenting the binary with additional instructions that

keep track of the current taint values. While the overhead is much less than that of a similar, previous tool, LIFT incurs an average performance slowdown of 3.6X for SPEC-INT [32].

2.2.4.3. Pointer Encryption

A novel protection scheme involves encrypting return addresses and function pointers when they are stored then decrypting them when they are loaded [40]. Since every return address goes through a decryption before it is loaded into program counter, a typical stack smashing attack would result in execution jumping to a random location in memory. This is because the address of the shell code (presumably stored within the overflowed buffer) would be decrypted and result in an unknown value, but most likely not where the attack desired. In the original paper, a variety of methods are offered to minimize the encryption overhead and ensure cryptographic security.

2.2.4.4. Shadow Threads

A recent innovation takes advantage of evolving multicore architectures by spawning a “shadow thread” to ensure security of the executing main thread [4]. Ideally, each thread would be running on a separate core thereby making use of idle cores. First the binary is modified to allocate “shadow memory” which will be used to indicate the level of trust associated with a piece of data. As the main thread executes regular program instructions, the shadow thread stays a few instructions behind and performs regular tainting arithmetic. The two threads use a synchronizer which relays control flow information from the main thread to the

shadow thread. Before branches are taken, the main thread waits for the shadow thread to catch up and then evaluates whether the branch target is safe or not. The shadow thread serves as a way to get the extra tainting functionality without adding additional code to the program or introducing new memory architectures. Runtime performance varied from 1.5X – 5.5X slowdown on a suite of custom applications and one program from SPEC-INT [4].

2.2.5. Operating System Patches

PaX is a Linux kernel patch written by The PaX Team whose principal author chooses to remain anonymous. PaX's main avenues of defense are to mark data as non-executable and take advantage of address space layout randomization (ASLR) [34]. By default, PaX marks the following areas as non-executable: memory that holds the stack, heap, anonymous memory mappings, and any section not specifically marked as executable in an ELF file. This prevents the standard stack-smashing attack since shell code stored in the buffer on the stack will be marked as non-executable. PaX randomizes the location of the stack, heap, loaded libraries, and executable binaries thereby greatly reducing the likelihood of success for attacks that rely on hard coded addresses, such as a standard ret-libC attack. This protection, when combined with a hardware protection scheme such as the NX bit provides for powerful protection. It should also be noted that a successful attack on PaX has been published on Phrack [27]. It directly calls the dynamic linker's symbol resolution procedure to get around the ASLR aspect of Pax and then uses a traditional ret-libC exploit.

2.2.6. Hardware Based Solutions

Another class of buffer overflow prevention techniques are those that make modifications to the actual hardware of a CPU. By doing so, users would benefit from these additional features upon buying a new system and most likely would not even know they were there. Users wouldn't have to recompile the programs they have, download new ones, or manually patch their OS. Most hardware based solutions use the notion of "tainting" memory locations. These taint bits are either all stored together in one continuous piece of memory or are tacked on to the end of every memory location. The latter is the more common case and has the advantage of being able to move and operate on taint bits with each instruction. In this particular scenario, all registers, caches, cache lines, and main memory would have to be widened by some amount. The operating system is expected to identify unsafe channels of input and mark the corresponding data as it streams into the computer. This change to the OS can be expected with the new hardware that the user buys.

2.2.6.1. Secure Bit2

Secure Bit2 extends every memory word and register by one bit which is used to add semantic meaning to each word of memory [31]. This bit is moved along with its associated word by memory manipulating instructions. Words in buffers passed between processes get their secure bit set while all others mark the secure bit at the destination register or memory location. Call, return, and jump instructions check the secure bit and if set, generate an interrupt or fault signal.

Modifications are required at the kernel level to set the secure bit when passing a buffer across domains. Since the address validation is done in hardware, there is no performance overhead. Memory overhead is related to the total size of memory, for *n* words of memory, an additional *n* bits are needed.

2.2.6.2. Raksha

Raksha offers multiple active security policies that can all be run simultaneously and are also programmable [11]. In this setup, every word of memory is extended by 4 bits, one for each security policy running. Each processor instruction carries out some operation for each of the security policies, one of which is solely devoted to high level attacks like SQL injection. This is what sets Raksha apart from many other hardware based solutions: the ability to recognize this type of sophisticated attack. In addition, software can modify the tag rules for each policy and configure how tags from multiple operands are combined. Raksha even allows a user to specify custom rules for a small number of individual instructions. These modifications were made to the open source Leon SPARC V8 processor and synthesized to a FPGA. The additional hardware caused a 7% increase in size over an unmodified Leon. Performance slowdown on SPEC ranged from 1X – 2.98X [11].

2.2.6.3. Minos

While Raksha is one of the newer tainting architectures, Minos is one of the older ones **Error! Reference source not found.**. Like Secure Bit, Minos extends each 32 bit word with a single integrity bit. The security policy can be

summarized as follows: any subject may modify any object if the object's integrity is not greater than that of the subject, but any subject that reads an object has its integrity lowered to the minimum of the object's integrity and its own. These operations are carried out in parallel with the normal functionality of the instruction. Minos is implemented and tested on the Pentium emulator known as Bochs. When an attempt is made to transfer control flow to low integrity data, the processor traps to the kernel for error recovery. A similar architecture is offered by Chen, et. al. in [42].

2.2.6.4. DIFT

Dynamic Information Flow Tracking (DIFT) also features modifications to the standard memory model by storing an extra bit for each byte of memory [36]. The memory overhead was greatly reduced by using multi-granularity tags. Each page of memory contains two extra bits which are used to indicate if all the taint bits on that page are the same and if they are all marked as trusted or not. If needed, the processor generates an exception to allocate more memory for individual bytes on a page, but by default marks all taint bits the same. This method reduced total memory overhead from 12.5% to 0.21% due to common occurrence of entire pages of memory having the same tag. Despite being publicized in 2004, DIFT remains the only tainting scheme that makes use of multi granularity tags. Two different policies are offered depending on the amount of performance one is willing to sacrifice in exchange for security. This

approach also uses binary annotation to recognize when data is properly bounds checked. When this occurs, the destination buffer can be marked as trusted.

2.3. Summary

This chapter explores some of the past attempts to eliminate memory based vulnerabilities dating back to 1959. Recent efforts up to 2007 have also been examined. Ranging from requiring developers to code in an entirely different language to coming built into new computer designs, they provide differing levels of security based on the amount of performance sacrificed or user involvement necessary. By examining the advantages and disadvantages of past solutions, a novel computer architecture that protects against buffer overflows is offered.

3. Approach

After thoroughly studying past and present protection schemes as well as modern attacks, the goal of this thesis is to provide a solution that fulfills the following requirements:

- Provides protection from known memory based vulnerabilities such as buffer overflows
- Works with existing legacy code
- Does not require recompilation
- Is compatible with emerging trends in processor architectures (multicore)
- Gives the maximum amount of security while sacrificing the least amount of performance
- Requires the least amount of user intervention
- Is mandatory as opposed to discretionary

3.1. Design Details

Working from these goals towards a specific solution leads one to believe that the correct approach is one that is hardware based. By employing a hardware based protection scheme, the user gains a security advantage without having to install new software or recompile their existing software. Whenever the user buys their new system, the protections are built in.

3.1.1. Memory extension

3.1.1.1. Byte or Word?

In the past, some tainting architectures have chosen to add one bit per byte [11] [36] [32] [42] [21] while others have chose to taint per 32 bit word [31] [9]. This approach will follow the former, more popular scheme. That is, every addressable memory location remains 8 bits wide but also has a one bit tag at the end. So each 4 byte word will have 32 bits of data along with 4 one bit tags interspersed throughout its bytes. Many processors, particularly x86, allow byte granularity memory access. Even though compilers normally align memory accesses to words or double words, there are still many cases in C that make word level tainting impractical. Aggregate types such as unions are one particular problem because they are often accessed in different ways, even without modifying data. On the other hand, a difficulty that arises from tainting at the byte level comes when data is accessed in large chunks that contain separate taint values. For example, imagine a union that can be accessed as 4 chars or 1 integer. If the chars were stored separately, they could each have a different taint value. Which does the CPU use in computing the taint value for the destination? In this work, the taint values are ORed together so that the result is the least trusted of all the bytes.

3.1.1.2. Architecture

The taint bit extension has to be made to every single memory location including registers, caches, and main memory. An abstract view of this CPU is shown in

Figure 3.1. The additional hardware is shown by a darker shade. This particular CPU is a four core system that is composed of 2 dual core packages. The diagram is based on the Intel Core 2 Quad architecture codenamed Kentsfield with model numbers in the Q6xx0 range [17]. Each core has its own L1 instruction and data cache and shares a unified L2 cache with the other core on the package. The taint unit of the L1 instruction cache must be wired to the ALU for verification in the case of a branching instruction. The tainting ALU (TALU) enforces the taint propagation rules for combining tainted and untainted data. The registers also have been extended to accommodate the taint bits and forward their status on to the TALU as well. While at first the wiring complexity may seem staggering, one should keep in mind that Figure 3.1 is just an abstract view of what the processor would look like. In reality, taint bits wouldn't be kept in a separate space on the same level of memory. Instead, they would be kept right beside the data as if each memory location were 9 bits wide. Adding one extra wire for each byte of data should not drastically complicate the wiring of a modern CPU. An extensive study of the area required to implement tagging bits is made in [9] and is estimated to be less than 0.5% for per-word tagging. Even with per byte tagging, the cost will be minimal compared to the size of modern x86 processors.

3.1.1.3. Overhead

Memory overhead for byte level tainting is 12.5%. A sample stack with taint extensions can be seen in Figure 3.2.

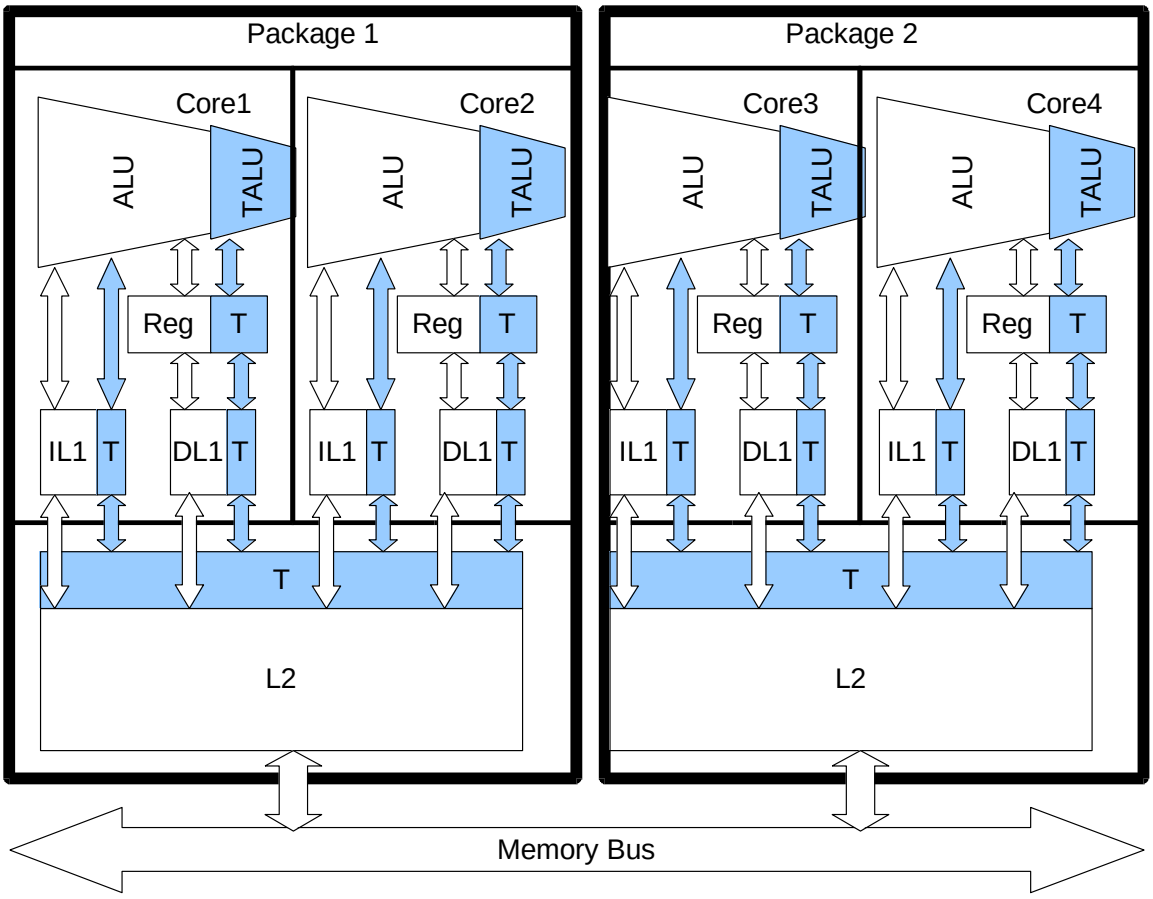

Figure 3.1: A 4 Core Processor with Tainting Hardware

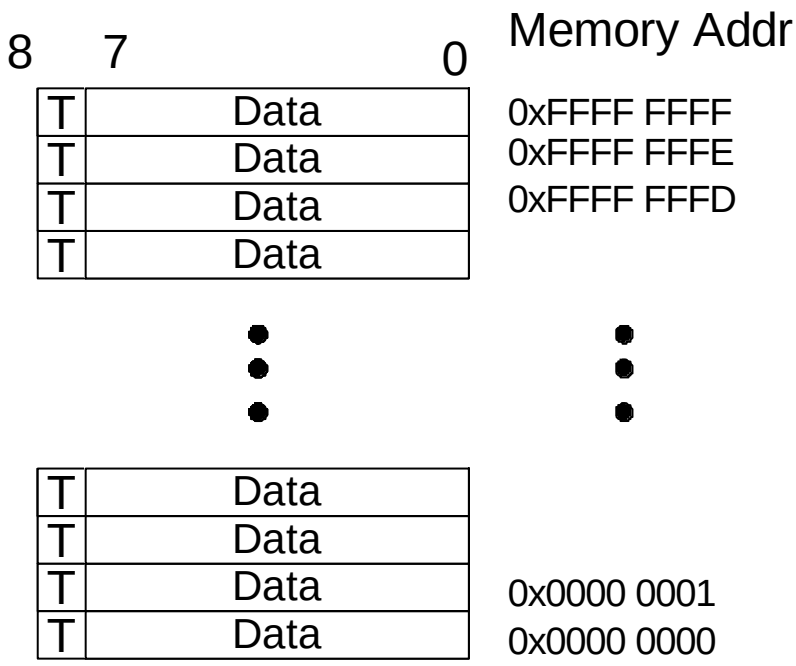

Figure 3.2: Memory layout of proposed architecture.

This is a small price to pay for additional security. At the current rate that memory densities are growing, this price can be paid in a very short time by Moore's Law.

All memory and cache lines will also have to be extended to facilitate transporting the extra information along the data path. Taint bits are passed through the memory hierarchy from registers to L1, L2, or main memory right alongside the actual data.

3.1.2. Taint Propagation Rules

Most importantly, a set of rules must be devised for how taint bits interact with each other. From a security standpoint, most of these rules are pretty straightforward. For example, the taint value for the result of an addition operation should be the least trusted of the two operands. For this application, a taint value of "1" means that the data stored at that location is not trusted. Conversely, a taint value of "0" means that that particular piece of memory is safe to branch to. A set of taint propagation rules is listed in Table 3.1. These rules are a modified set based on the rules in [11] [36] [32] [42] [21]. Note that ALU represents typical ALU operations such as ADD, SUB, OR. The special case XOR entry in the table corresponds to a commonly used compiler technique to clear a register. In doing so, the destination register will be all zeros and can be considered to be trusted.

Table 3.1: Taint Propagation Rules.

Instruction	Meaning	Rules
ALU R1, R2, R3	R1 <- R2 + R3	T[R1] <- T[R2] OR T[R3]
LW R1 IMM(R2)	R1 <- Mem[R2+IMM]	T[R1] <- T[R2] OR T[IMM]
SW R1, IMM(R2)	Mem[R2+IMM] <- R1	T[R2+IMM] <- T[R1] OR T[IMM]
XOR R1, R2, R2	R1 <- R2 XOR R2 (R1 <- 0)	T[R1] <- 0

3.1.3. Additional ALU functionality

Additionally, the CPU is modified to propagate taint information while performing normal operations. Typical assembly commands like add or load now contain extra functionality. This extra functionality comes from the ALU itself, not instructions that are annotated to the binary. This allows legacy code to run without having to recompile or instrument the executable. Given the scale and complexity of modern chips, the additional space required to carry out the taint propagation rules will be minimal because the operators are so simple. Branch instructions like RET, CALL, or JMP must validate the taint bits before branching to that location.

3.1.4. Floating Point, MMX and SSE

Since floating point data should never be used as the target for a branch or return instruction, all floating point data is marked as tainted. Memory extensions are not needed by the FPU or FP registers because when writing floating point data back to memory, it is automatically marked as untrusted.

Multimedia / vector instructions such as MMX and SSE and all their variations can be considered in the same way. No extra functionality is performed by these instructions because they operate on floating point data and as such are forced to write back to memory with low integrity.

3.1.5. Reaction to Attack

When a branching instruction is found to be tainted, an attack is signaled. Upon discovery, there are a variety of options that could be taken: trap to the operating system, terminate the process, or attempt to recover. For the time being, whenever an attack is found, the application is terminated. Repeated attack attempts could lead to a Denial of Service but this a better result than a hacker hijacking control flow.

3.2. Bochs

The proof of concept for this work is done using an IA-32 emulator called Bochs [22]. Bochs is an open source C / C++ project maintained on SourceForge. Within the computer architecture community, it is a commonly used tool to verify designs, particularly experimental ones [31] [9] [40]. By using Bochs the developer gets a chance to quickly see the results of changes made to the internals of a CPU. Modifications can be tested by running different applications on their new, unique design. Bochs uses a custom BIOS and can emulate a standard PC including memory, DMA, I/O devices, and an x86 CPU with MMX and SSE instructions. A variety of processor cores can be emulated ranging from x386 to P4.

Any machine with a C++ compiler can run Bochs provided they have the correct display libraries. The ability to run on a variety of host operating systems while emulating a machine running a different OS makes Bochs useful for virtualization or running Linux programs natively on a Windows machine or vice versa.

Figure 3.3 shows a screenshot of Bochs using Windows XP as its host OS. Inside the emulator, Bochs is running Redhat 6.0. Some other tested guest operating systems are FreeDOS [14], openBSD [29], and nearly every flavor of Windows including Vista [25]. By downloading the Bochs source code and making changes to it one can easily experiment with the internal design of CPU. A snippet of the code used to emulate the `ADD EAX, Immediate` instruction is shown in Figure 3.4. If, for example, a printf() statement were entered in the function then every time that the assembly instruction was executed Bochs would print a statement to the terminal in Windows. A more practical use would be to add some sort of extra functionality to the add instruction. One such use would be propagating taints bits. Being written in C makes Bochs much more accessible and easier to use than other hardware simulators. At program startup, extra storage area for taint bits had to be allocated. Each and every integer ALU instruction had to be modified to include taint bit propagation as dictated by the tainting rules in Table 3.1.

3.2.1. Implementation

To facilitate testing of this new processor design, Bochs was modified to include taint bits for each memory location. This involved slight modifications to many Bochs functions. Dissecting and understanding the interactions and overall functionality of an open source program such as Bochs proved to be quite challenging. Memory load and store instructions had to be modified to store the correct taint bits after doing the lookup from virtual address to physical address.

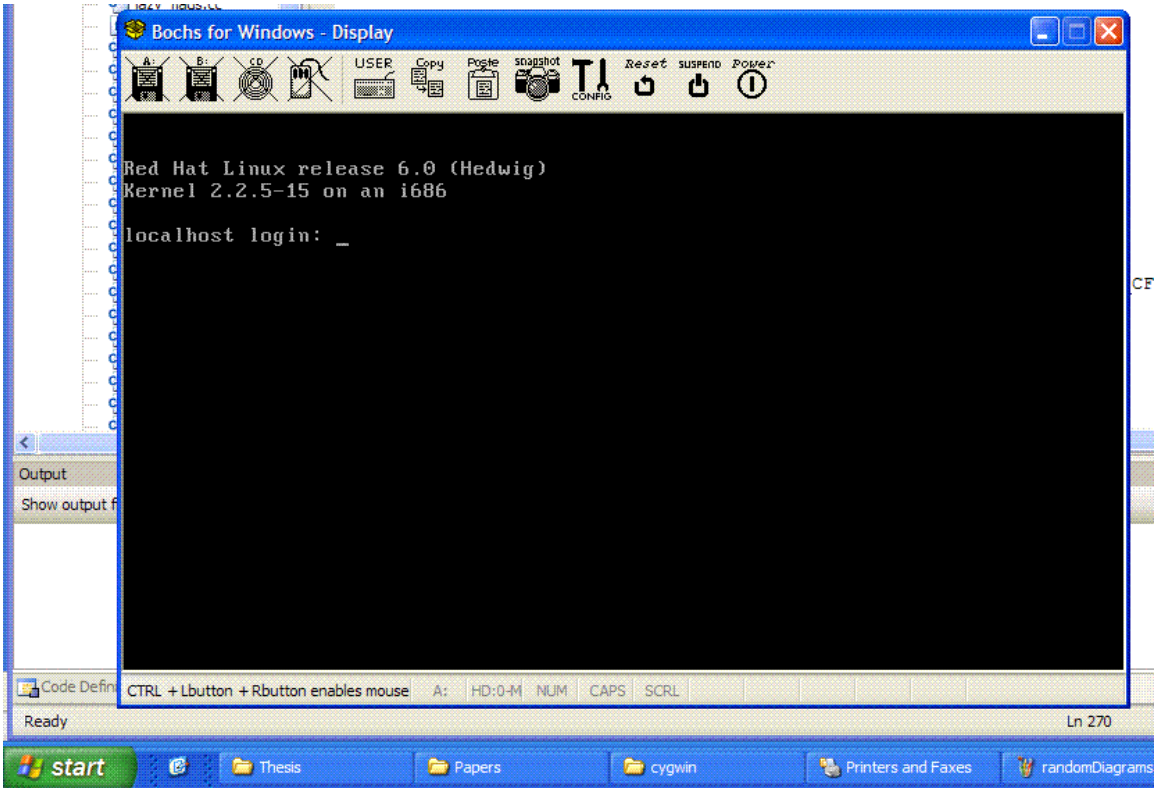

Figure 3.3: Redhat Linux running under Windows XP through the Bochs Emulator

```
void BX_CPU_C::ADD_EAXId(bxInstruction_c *i)
{
  int op1_32, sum_32;
  op2_32 = i->Id();

  op1_32 = EAX;
  sum_32 = op1_32 + op2_32;
  RAX = sum_32;

  SET_FLAGS_OSZAPC_ADD_32(op1_32, op2_32,
sum_32);
}
```

Figure 3.4: Bochs Code Snippet for Add EAX, Imm instruction

The ALU had to be modified to not branch to addresses that were marked as insecure. If the target of a branch instruction was untrusted, the program is terminated. Before the performance and security of the overall system could be tested, the modified version of Bochs had to successfully boot a Linux disk image. Upon doing so, actual testing to verify Boch's ability to repel attacks could begin.

3.3. Summary

In this chapter, the idea of extending memory to include taint bits on a per byte basis was presented. A logical overview of how a modified system would look was provided. To go along with this depiction, some estimations were made concerning die size and memory overhead required to produce processors of this type. A set of rules was laid down describing the interactions between taint bits depending on the currently executing instruction. Bochs, the platform for the proof of concept of this work, was described along with a description of the changes made to it. The next chapter will describe the performance of this modified CPU.

4. Results and Discussion

To test and verify the proposed architecture, an attack was constructed then tested on an unmodified version of Redhat 6.0 running under Bochs. Being able to show a successful attack before modifications and a repelled attack with protections in place is adequate to demonstrate a security advantage.

4.1. The Attack

A buffer overflow style attack was written using [2] as a guide. The attack comes in two parts: an exploit program that constructs the malicious string and a program that has a vulnerable strcpy() call.

4.1.1. Exploit Program

The purpose of the first program is to form a malicious string that can then be used to overflow another program's buffer. The application takes a buffer size and an offset from its own stack pointer as parameters to create the string. With these numbers, the program creates an "egg" which is a malicious string that consists of a NOP sled, shell code, and return addresses. The NOP sled is placed at the bottom of the egg to allow for some error in the return address. If the return address returns anywhere in the NOP sled then an unspecified number of NOPs is run until the shell code is reached. After determining the correct length of the buffer to overflow, the length of the NOP sled and return address region are calculated based on the amount of shell code which will be put in the buffer.

A memory layout of the egg is given Figure 4.1. This egg is stored in an environment variable for easier access. The purpose of the code in the buffer is to open a shell by using the system() command. When paired with a poorly coded vulnerable program, this combination makes for a particularly dangerous attack.

4.1.2. Vulnerable Program

The program which will be attacked is a simple rootecho program. On a nonprivileged account it uses setuid() to run at root privilege. This characteristic makes the program particularly enticing to exploit because the attacker can gain root access if the exploit is done correctly. The intended purpose of the program is to echo the first command line argument given with it. The code for this program can be found in Figure 4.2.

4.1.3. A Successful Attack

A successful attack involves a number of different things. First, the exploit program must be run to create the malicious string. Knowing the approximate size of the target buffer is greatly helpful in this process. The popularity and prevalence of open source software makes this much easier than it may have been in the past. Open source software also helps hackers to identify vulnerable programs. By running grep and examining locations where unsafe function calls are made, an attacker can easily find a suitable target. Second, the vulnerable program needs to be run using the malicious string as input. Lastly, it helps to be a little lucky since stack addresses can vary from execution to execution.

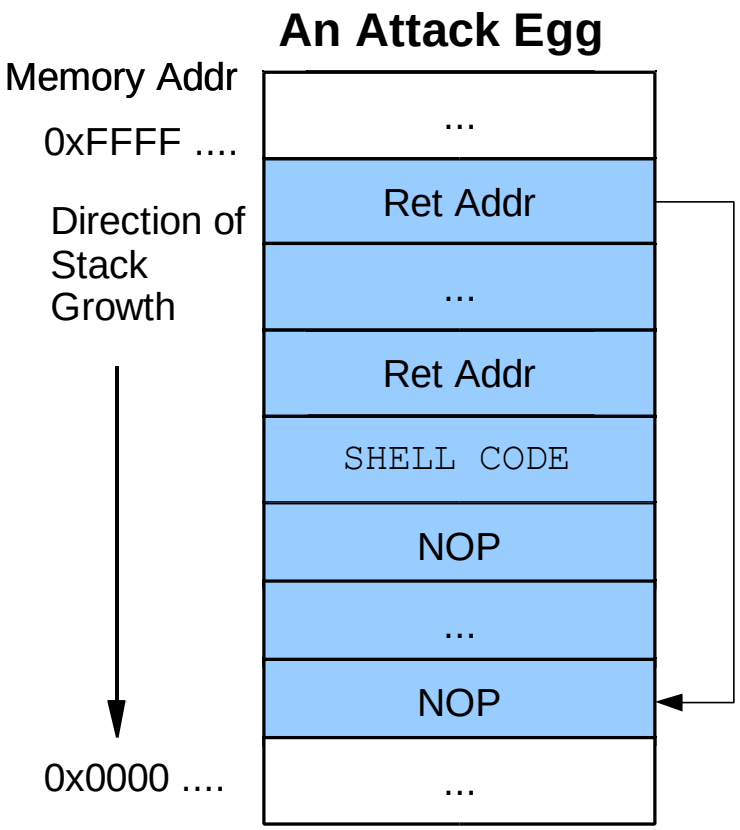

Figure 4.1: An egg that would be used to overflow a buffer.

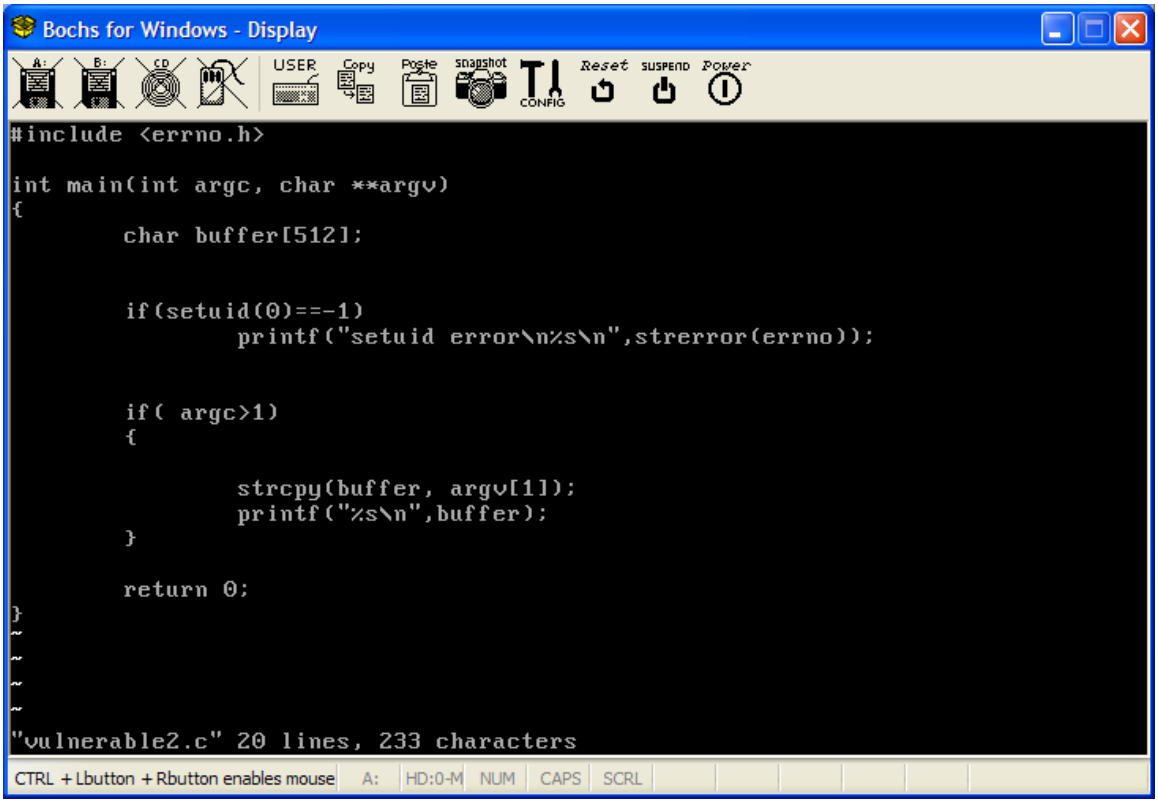

Figure 4.2: Rootecho program.

However, the NOP sled goes a long ways to reduce the chances of returning into the wrong address. A successful attack showing an elevation to root privilege is shown in Figure 4.3. A portion of the string is printed to the screen before a root shell is given.

4.2. Defense

The proposed solution effectively prevents the above attack from being successful. Stopping an attack from gaining root privileges is noteworthy and validates the approach taken in this work. Instead of the attacker having unfettered access to someone's machine, the result of an attack is a segmentation fault. As stated previously, the semantics of branching functions are modified to validate the taint bits associated with the destination. When this location is found to be tainted, the instruction is not executed and the program crashes. A screenshot of this is given in Figure 4.4. Again, a portion of the attack string is printed, but this time a segmentation fault is given instead of a root shell.

4.3. Discussion

The above results demonstrate the ability to prevent a typical stack smashing attack. While this particular example may seem simple, an unprotected system can be successfully exploited exactly this easily. As previously mentioned, the goal of this work was to provide a hardware based solution that would be helpful to the everyday while requiring little to no effort on their part.

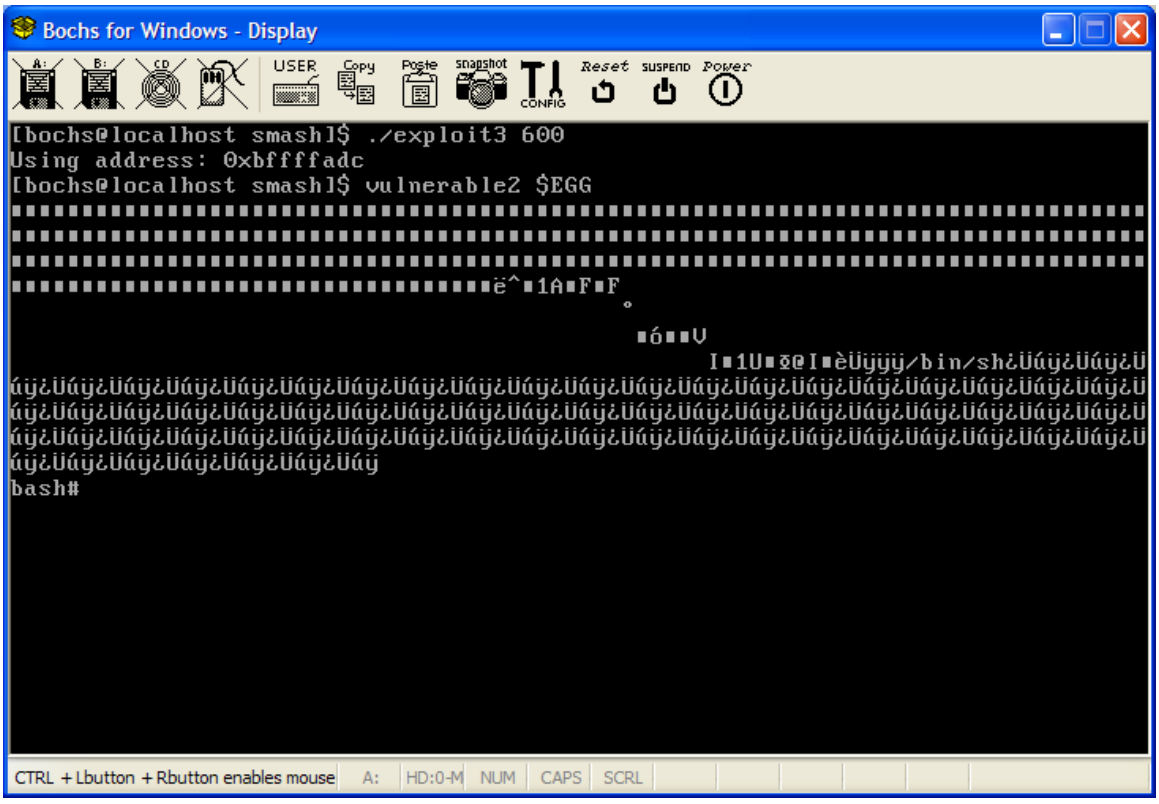

Figure 4.3: Successful attack.

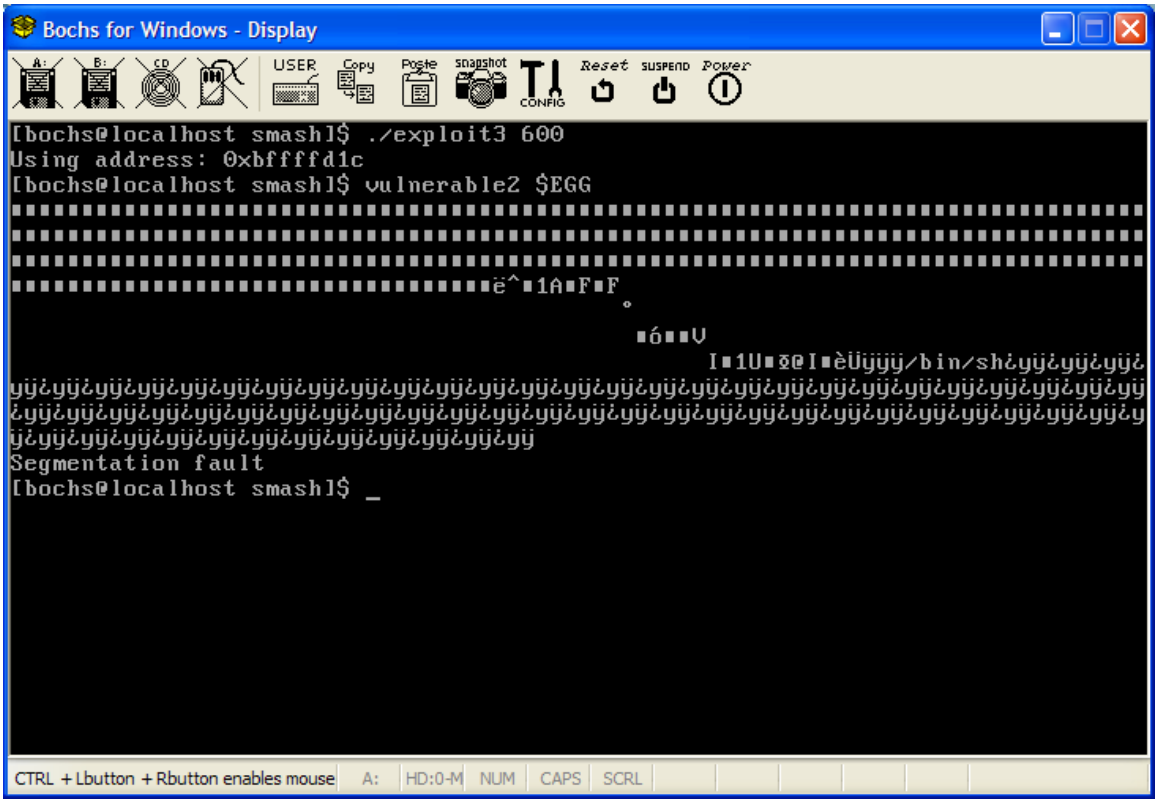

Figure 4.4: An unsuccessful attack

A fully functional and updated SELinux machine would not be susceptible to the demonstrated attack. At the same time, understanding and maintaining a SELinux computer is a challenging task in its own right. Failing to grasp all the nuisances and security policies of the operating system usually results in a number of applications being labeled insecure and thus, unusable. On the other side of the spectrum is the supposed user friendliness of Windows. While Windows may be the everyday user's operating system of choice, it is certainly not the most secure. The popularity of Windows among laymen has made it the most popular platform for attack by hackers because of the number of people using it insecurely. The lengths that one has to go to in order to keep their version of Windows up to date are substantial at times [38]. Because of this, some users may elect to disable automatic updates spurred in part by the frequent restarts required for an update to take effect. Presumably, this would lead to one of the more exposed systems imaginable: an out of date Windows machine.

4.4. Summary

This chapter detailed the exploit and vulnerable programs that were used to test the modified multicore CPU. A buffer overflow attack was used in combination with a program that used an unsafe call to strcpy() resulting in escalation of privileges for a nonroot user. A processor that features byte level tainting was able to repel this same attack under an identical test environment. A computer

featuring this type of processor would be immune to stack smashing attacks without ever requiring the user to install new software, download updates, or recompile existing software. The mandatory nature of this protection scheme would make circumventing it particularly difficult. All of this is accomplished while still being backwards compatible with legacy code.

5. Conclusions

Memory based vulnerabilities have been a serious problem over the past two decades. Over that time, a great deal of work has been done to overcome them. However, much of this work required a substantial amount of expertise and involvement from the user and was therefore unsuccessful. For this reason, exploits as simple as buffer overflows remain dangerous despite their age. The subject of this work has been to close off this attack vector while requiring the least amount of user intervention. To accomplish this goal, a hardware based mandatory access control mechanism was implemented. This allows for existing legacy code to be executed natively without recompilation. To be practical, the design was made with multicore consideration, an aspect of tainting that has yet to be researched. To demonstrate its ability to repel attacks, a pair of programs were created that result in a root shell for an unprivileged user. These programs were successfully exploited on a typical system using the x86 emulator Bochs. When the hardware MAC was implemented, the attacks failed. This was accomplished with only a 12.5% memory overhead and little to no performance degradation.

5.1. Future Work

As the system seems to show promise, there remains work to be done. At the forefront of this list would be an actual VHDL implementation that could then be run on a FPGA. Many FPGAs already feature hard processors and can run Linux. Creating a custom data path that features this hardware MAC would be a

challenge worth the time and effort. Unfortunately, this requires the VHDL code for an actual processor. Since the platform of choice for this work was x86, the code could be difficult to come by. If the platform were not important, there is a popular SPARC open source core available called Leon. Instantiating multiple cores on an FPGA and then recreating these attacks would be a major step.

In addition to the next steps in implementation, a variety of new attacks should be tested also. The number of attacks that specifically target multicore architectures are few but growing. Capturing one of these attacks in the wild and then testing it on Bochs or an FPGA would be productive work as well.

Along the way, it is expected that the design would be continuously updated as well. Having a graceful return from a detected attack would be favorable compared to a segmentation fault. In the latter case, repeated attacks could result in a Denial of Service (DoS). For some companies such as Amazon, a successful DoS attack can be very damaging.

Another goal is to recognize properly bounds checked data and untaint it. Previous tainting schemes have attempted this with varying degrees of success. The difficulty is recognizing a set of instructions that qualifies as a bounds check. Moving the solution up to the compiler level would solve this problem, but require source code and recompilation.

All of these areas deserve future consideration but at the time being, the solution serves its purpose in defending against buffer overflow style attacks.

List of References

List of References

[1] "A detailed description of the Data Execution Prevention (DEP) in Windows XP Service Pack 2, Windows XP Table PC Edition 2005, and Windows Server 2003". http://support.microsoft.com/kb/875352. Retrieved 2 July 2008.

[2] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1998.

[3] Borin, E., Wang, C., Wu. Y., Araujo, G. "Software-Based Transparent and Comprehensive Control-Flow Error Detection". *Proceedings of the International Symposium on Code Generation and Optimization*, p.333-345, March 26-29, 2006 [doi>10.1109/CGO.2006.33]

[4] Chabbi, M. M. "Efficient taint analysis using multicore machines," Master's thesis, The University OF Arizona, 2007.

[5] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. 2005. "Non-control-data attacks are realistic threats". In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Baltimore, MD, July 31 - August 05, 2005). USENIX Association, Berkeley, CA, 12-12.

[6] Cowan, C. "Stackguarded Red Hat 5.2 Released". https://honor.icsalabs.com/pipermail/firewall-wizards/1999-October/006584.html Retrieved 1 July 2008.

[7] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference* (San Antonio, TX, Jan 1998). 63-78.

[8] Cowan, C., Wagle, P., Pu, C., Beattie, S., and Walpole, J. 2000b. "Buffer overflows: Attacks and defenses for the vulnerability of the decade". In *DARPA Information Survivability Conference and Exposition* (DISCEX 2000). 119--129.

[9] Crandall, J. R. and Chong, F. T. 2004. "Minos: Control Data Attack Prevention Orthogonal to Memory Model". In *Proceedings of the 37th Annual IEEE/ACM international Symposium on Microarchitecture* (Portland, Oregon, December 04 - 08, 2004). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 221-232. DOI= http://dx.doi.org/10.1109/MICRO.2004.26

[10] CVE Statistics. Retrieved 24 June 2008. www.nvd.nist.gov/

[11] Dalton, M., Kannan, H., and Kozyrakis, C. 2007. "Raksha: a flexible information flow architecture for software security". *SIGARCH Computer Architecture News* 35, 2 (Jun. 2007), 482-493. DOI= http://doi.acm.org/10.1145/1273440.1250722

[12] Dean, D., Felten, and Wallach 1996. "Java Security: From HotJava to Netscape and Beyond". In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (May 06 - 08, 1996). SP. IEEE Computer Society, Washington, DC, 190.

[13] "Execute Disable Bit Functionality Blocks Malware Code Execution". Accessed 11 September 2007. http://cache-www.intel.com/cd/00/00/14/93/149307_149307.pdf

[14] "Free DOS: The FreeDOS Project". http://www.freedos.org/. Retrieved 1 July 2008.

[15] Ghory, Z. "Protecting Systems with Libsafe". *Security Focus.* August 20, 2001. http://www.securityfocus.com/infocus/1412

[16] Hennessy, John L and Patterson, David A. *Computer Architecture: A Quantitative Approach*, 4th ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, 2007

[17] "Intel Core 2 Quad Processor Overview." http://www.intel.com/products/processor/core2quad/index.htm. Retrieved 2 July 2008.

[18] Houdek, M. E., Soltis, F. G., and Hoffman, R. L. 1981. "IBM System/38 support for capability-based addressing". In *Proceedings of the 8th Annual Symposium on Computer Architecture* (Minneapolis, Minnesota, United States, May 12 - 14, 1981). International Symposium on Computer Architecture. IEEE Computer Society Press, Los Alamitos, CA, 341-348.

[19] Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang Y., "Cyclone: A Safe Dialect of C", *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, p.275-288, June 10-15, 2002

[20] Kiriansky, V., Bruening, D., and Amarasinghe, S. P. 2002. "Secure Execution via Program Shepherding". In *Proceedings of the 11th USENIX Security Symposium* (August 05 - 09, 2002). D. Boneh, Ed. USENIX Association, Berkeley, CA, 191-206.

[21] Kong, J., Zou, C. C., and Zhou, H. 2006. "Improving Software Security via Runtime Instruction-Level Taint Checking". In *Proceedings of the 1st Workshop on Architectural and System Support For Improving Software Dependability* (San Jose, California, October 21 - 21, 2006). ASID '06. ACM, New York, NY, 18-24. DOI= http://doi.acm.org/10.1145/1181309.1181313

[22] Lawton, K. P. 1996. "Bochs: A Portable PC Emulator for Unix/X". *Linux Journal.* 1996, 29es (Sep. 1996), 7.

[23] Levy, H. M. 1984 *Capability-Based Computer Systems*. Butterworth-Heinemann.

[24] Mayer, A. J. 1982. "The architecture of the Burroughs B5000: 20 years later and still ahead of the times?". *SIGARCH Computer Architecture News* 10, 4 (Jun. 1982), 3-10. DOI= http://doi.acm.org/10.1145/641542.641543

[25] "Microsoft Corporation". http://www.microsoft.com/. Retrieved 1 July 2008.

[26] Moore, D., Shannon, C., and Claffy, k. 2002. "Code-Red: a case study on the spread and victims of an internet worm". In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement* (Marseille, France,

November 06 - 08, 2002). IMW '02. ACM Press, New York, NY, 273-284. DOI= http://doi.acm.org/10.1145/637201.637244

[27] Nergal. "The advanced return-into-lib(c) exploits: PaX case study". *Phrack Magazine* 58(4), December 2001. http://www.phrack.org/issues.html?issue=58&id=4#article

[28] Nologin.org. "Bypassing Windows Hardware-Enforced Data Execution Prevention". Accessed 11 September 2007. http://uninformed.org/?v=2&a=4&t=pdf

[29] "OpenBSD". http://www.openBSD.org/. Retrieved 1 July 2008.

[30] Orman, H. "The Morris Worm: A Fifteen-Year Perspective". *IEEE Security & Privacy Magazine, 1,* 5 (Sept. – Oct. 2003), 35-43.

[31] Piromsopa, K., and Enbody, R. 2005. "Secure Bit2: Transparent, Hardware Buffer-Overflow Protection". Technical Reports #MSU-CSE-05-9, Department of Computer Science and Engineering, Michigan State University (2005).

[32] Qin, F., Wang, C., Li, Z., Kim, H., Zhou, Y., and Wu, Y. 2006. "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks". In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (December 09 - 13, 2006). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 135-148. DOI= http://dx.doi.org/10.1109/MICRO.2006.29

[33] Ruwase, O. and Lam, M. "A practical dynamic buffer overflow detector". In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, Feb 2004.

[34] Silberman, P., and Johnson, R. "A Comparison of Buffer Overflow Prevention Implementations and Weaknesses", I-Defense, 1875 Campus Commons Dr. Suite 210 Reston, VA 20191, http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf

[35] Smalley, S., Vance, C., and Salamon, W. "Implementing SELinux as a Linux security module". NAI Labs Report #01-043, NAI Labs, Dec 2001. Revised May 2002.

[36] Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S. 2004. "Secure Program Execution via Dynamic Information Flow Tracking". In *Proceedings of the 11th international Conference on Architectural Support For Programming Languages and Operating Systems* (Boston, MA, USA, October 07 - 13, 2004). ASPLOS-XI. ACM Press, New York, NY, 85-96. DOI= http://doi.acm.org/10.1145/1024393.1024404

[37] "Study: PEBKAC still a serious problem when it comes to PC security". Retrieved 24 June 2008. http://arstechnica.com/news.ars/post/20071001-study-pebkac-still-a-serious-problem-when-it-comes-to-pc-security.html

[38] "TCO for Application Servers: Comparing Linux with Windows and Solaris". Robert Frances Group. August 2005 120 Post Road West, Suite 201, Westport, CT 06880.

[39] "TIOBE Programming Community Index for July 2008". TIOBE Software. Retrieved 31 July 2008. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[40] Tuck, N., Calder, B., and Varghese, G. 2004. "Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow". In *Proceedings of the 37th Annual IEEE/ACM international Symposium on Microarchitecture* (Portland, Oregon, December 04 - 08, 2004). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 209-220. DOI= http://dx.doi.org/10.1109/MICRO.2004.20

[41] "Unisys Mainframes: Secure MCP Systems – Secure by Design". Accessed 2 September 2007. http://www.unisys.com/products/mainframes/security/secure__mcp__systems/secure__by__design.htm

[42] Xu, J. and Nakka, N. 2005. "Defeating Memory Corruption Attacks via Pointer Taintedness Detection". In *Proceedings of the 2005 international Conference on Dependable Systems and Networks* (June 28 - July 01, 2005). DSN. IEEE Computer Society, Washington, DC, 378-387. DOI= http://dx.doi.org/10.1109/DSN.2005.36

[43] Zeichick, A. "Security Ahoy! Flying the NX Flag on Windows and AMD64 To Stop Attacks". http://developer.amd.com/documentation/articles/pages/3312005143.aspx , Retrieved 1 July 2008.

[44] Zhang, K., Zhang, T., and Pande, S. 2006. "Memory Protection through Dynamic Access Control". In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (December 09 - 13, 2006). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 123-134. DOI= http://dx.doi.org/10.1109/MICRO.2006.33

Appendix A – exploit.c

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define NOP                            0x90

char shellcode[] =

"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x
b0\x0b"

"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x
40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
    buff[i] = NOP;

  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
  for (i = 0; i < strlen(shellcode); i++)
```

```
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Vita

Brian Sharp was born in Dallas, TX on October 21, 1984. Shortly thereafter, he moved to Powell, TN, a suburb of Knoxville. He attended public schools there and received a diploma in May 2003. Entering the University of Tennessee, Knoxville, that fall, he would later graduate in May 2007 with a Bachelors of Science in Computer Engineering. During his undergraduate course work, he became interested in Computer Architecture, Embedded Systems, and Digital Design. Encouraged by a summer internship at the Air Force Research Lab in Rome, NY, Brian pursued a Masters Degree in Computer Engineering at the University of Tennessee, Knoxville. He would go on to graduate in August 2008.