A
Dissertation
On

PRIORITIZED RUNTIME ADDRESS RANDOMIZATION FOR BUFFER OVERFLOW EXTENUATION

Submitted in Partial fulfillment of the requirement for the Award of the Degree of

MASTER OF ENGINEERING
In
Computer Technology and Applications

Submitted by
Rahul Kumar Agrawal
College Roll No. 14/CTA/06
University Roll No. 10193

Under the Guidance of
Dr. DAYA GUPTA
(HOD)
Department of Computer Engineering

DEPARTMENT OF COMPUTER ENGINEERING
DELHI COLLEGE OF ENGINEERING
BAWANA ROAD, DELHI -110042
(UNIVERSITY OF DELHI)

2006-2008

CERTIFICATE

DELHI COLLEGE OF ENGINEERING

(Govt. of National Capital Territory of Delhi)

BAWANA ROAD, DELHI - 110042

Date: _______________

This is certified that the major project report entitled "**Prioritized Runtime Address Randomization for Buffer Overflow Extenuation**" is the work of **Rahul Kumar Agrawal** (University Roll No. **10193**), a student of Delhi College of Engineering. This work was completed under my direct supervision and guidance and forms a part of the Master of Engineering (Computer Technology & Applications) course and curriculum. He has completed his work with utmost sincerity and diligence.

The work embodied in this dissertation has not been submitted for the award of any other degree to the best of my knowledge

Dr. DAYA GUPTA
Assistant Professor and HOD
Project Guide
Department of Computer Engineering
Delhi College of Engineering, Delhi

ACKNOWLEDGEMENT

It is a great pleasure to have the opportunity to extent my heartfelt gratitude to everybody who helped me throughout the course of this project.

It is distinct pleasure to express my deep sense of gratitude and indebtedness to my learned supervisor Dr. Daya Gupta for her invaluable guidance, encouragement and patient reviews. Her continuous inspiration has made me complete this dissertation. She kept on boosting me time and again for putting an extra ounce of effort to realize this work.

I would also like to take this opportunity to present my sincere regards to my faculty members Dr. S K Saxena, Mrs. Rajni Jindal, and Mr. Manoj Sethi for their support and encouragement.

I am grateful to my parents for their moral support all the time; they have been always cheering me up in the odd times of this work.

I am also thankful to my friends for their unconditional support and motivation during this work.

Rahul Kumar Agrawal
College Roll No. 14/CTA/06
University Roll No. 10193
M.E. (Computer Technology & Applications)
Department of Computer Engineering
Delhi College of Engineering, Delhi-42

ABSTRACT

Today a majority of security exploits have a memory corruption component. These attacks depend on corrupting some process memory and either injecting a shellcode and executing that or using some system calls with malicious parameters to harm victim machine and to replicate. The existing approaches either provide insufficient randomness or require source code modification. Insufficient randomness allows attacker to use Brute-Force attacks and source code modification doesn't seem to be viable solution for all the software available on the internet which are the biggest reservoir of the memory corruption attacks. In this paper we propose PriCryp, a prophylactic security technique that makes efficient and effective use of ASLR and ISR to provide prioritized cryptographic runtime code block randomization. Here we make key observation that system calls are almost always used in memory corruption attacks by the attackers. We make use of cryptographic algorithms to scramble data variables and security critical instructions which we say including system call instruction and other application binary interface (ABI) functions. We prioritize the code blocks according to the system calls within them. PriCryp intercepts the process before execution and disassembles it to identify code blocks of related instructions. These code blocks are then assigned priority according to the number of system calls within them. Based on this priority the code blocks are then randomized during runtime. PriCryp randomizes the very components of a process including Stacks, Heaps, Data segments and Code segments. Further randomizes the code segment internally.

Table of Contents

Chapter 1. Introduction ... 6

1.1 Background ... 6

1.2 Motivation ... 8

1.3 Objective ... 9

1.4 Proposed Work ... 9

1.5 Thesis Organization ... 10

Chapter 2. Memory Exploitation & Preclusion ... 11

2.1 Memory ill-treatment Techniques ... 11

2.1.1 Introduction ... 11

2.1.2 Exploiting Buffer Overruns ... 13

2.2 Thwarting Memory Corruption ... 20

2.2.1 StackGuard ... 21

2.2.2 Code Obfuscation ... 22

2.2.3 Absolute Address Randomization ... 23

2.2.4 Instruction Set Randomization ... 23

2.2.5 NX/XD Bit ... 24

2.2.6 Address Space Layout Randomization [17] ... 27

2.2.7 DAWSON [19]] ... 27

2.2.8 ASLP (Address Space Layout Permutation) [20] ... 28

2.2.9 RISE (Randomized Instruction Set Emulation) [27] ... 28
2.3 Conclusion ... 29
Chapter 3. PriCryp Model ... 31
3.1 Design Goals ... 31
3.2 PriCryp Design ... 33
3.2.1 Course Grained Randomizer ... 34
3.2.2 Disassembler ... 35
3.2.3 Prioritizing ... 37
3.2.4 Code Scrambler ... 37
3.2.5 Code Block Randomizer / Fine Grained Randomizer ... 39
3.2.6 Data Scrambler ... 40
3.3 Design Strengths ... 41
Chapter 4. Evaluation & Limitations ... 42
4.1 Evaluation ... 42
4.1.1 Unintentional Memory Corruption ... 42
4.1.2 Traditional Attacks ... 42
4.1.3 Chained Return-into-Libc Attacks ... 43
4.1.4 Brute Force Attacks ... 43
4.1.5 De-Randomization Attacks ... 44
4.2 PriCryp in Contrast with Others ... 44

4.3 Limitations 44

Chapter 5. Implementation Details 46

5.1 Tools Used 46

5.1.1 Visual C++ 6.0 46

5.1.2 OllyDBG 46

5.1.3 Dependency Walker 47

5.1.4 Metasploit Project 47

5.2 Bypassing DEP 47

5.3 Code 57

5.3.1 DEP.cpp 57

5.3.2 DEP1.c 57

5.3.3 DEP2.c 57

5.4 Screenshots 58

5.4.1 Bypassing Software Enforced DEP 58

5.4.2 Bypassing Hardware Enforced DEP 59

Chapter 6. Conclusion & Future Work 61

Chapter 7. Publications from the Thesis 63

7.1 Accepted Paper 63

7.2 Communicated Journal 64

Chapter 8. References 65

List of Figures

Figure 1: Structure of a Stack in a Process 15

Figure 2: General Structure of a Heap 19

Figure 3: Stack Smashing Attack........ 21

Figure 4: i386 Stack Layout with StackGuard........ 22

Figure 5: Return into libc Attack 24

Figure 6: PriCryp Components and the interface with the Process and the Kernel........ 32

Figure 7: Normal Process Memory Layout 33

Figure 8: PriCryp's program flow design 34

Figure 9: PriCryp Design Internals 35

Figure 10: Two Code Blocks Identified Using Disassembler 36

Figure 11: Randomized Code Blocks of figure 10 36

Figure 12: The return to libc attack string 48

Figure 13: Busy Block Header........ 50

Figure 14: Free Block Header........ 50

Figure 15: Lookaside Lists for Heap Management........ 52

Figure 16: Freelist Entries of free heap blocks 53

Figure 17: Lookaside List of heap blocks........ 53

Figure 18: Direction of Heap Overflow........ 54

Figure 19: Safe Unlink Operation........ 55

Figure 20: Free Block Header in Windows XP SP2 (Cookie Added)........ 55

Figure 21: Shellcode execution bypassing software enforced DEP 58

Figure 21: Shellcode execution bypassing hardware enforced DEP. 59

Figure 22: Popped Up Warning Message on Closing Windows Calculator..................... 60

List of Tables

Table 1: CERT Advisories (1999-2002)…………………………………………………..7

Table 2: Comparison of PriCryp and Other Techniques………………..……………….45

Chapter 1. Introduction

1.1 Background

Memory corruption exploits are most common vulnerabilities among the software vulnerabilities through which attacker can take control of the computers. More than 50 percent of today's widely exploited vulnerabilities are caused by buffer overflow and this percentage is increasing as time passes. An increasingly used attack is the Code Injection, to exploit the software vulnerability on the victim machine and execute either the remotely injected code or an existing malicious code. The worms like *Sasser* [1], *Blaster* [2], *Code Red* [3], and *Code Red II* use this attack to execute the injected code on the victim to infect and replicate. The *Code Red*, *Code Red II* and their variations exploited the known vulnerability in the Microsoft Index Service DLL. In 2003, *Sapphire* [4](SQL Slammer), and *MSBlaster* [2] took advantage of buffer overflow vulnerabilities to inject into systems. As attributed in the US-CERT Cyber Security Advisories [5] of recent years, a majority of security exploits are based on memory corruption component. In 2003, buffer overflow accounted for 70.4% (19 of 27) of the serious vulnerability reports from CERT advisories.

These memory corruption vulnerabilities are usually caused by insecure programming style and unavailability of secure libraries in the programming languages like C. These vulnerabilities are mostly occurred due to the lack of input validation in the C programming language, due to which programmers are free to decide when and how to tackle the input. However these programming styles sometimes result in improved performance of the application. Again the difficulty in source code analysis forces

programmers to make assumptions or simplification that causes a huge number of false positives or false negatives. The techniques for exploiting memory corruption vulnerabilities depend on the memory layout of the executing program under target. These vulnerabilities include Stack and Heap Overflows, and Underflows, Format Strings, Array Index overflows, and uninitialized variables. Typically these attacks are designed to enable the remote attacker to execute some arbitrary code on the victim remote machine. This execution may provide the attacker the full control of the victim system or may be used to propagate worms, data corruption or some hidden installation. These factors have fueled a lot of research into defenses against exploitation of memory corruption vulnerabilities. Early research targeted specific exploit types such as stack smashing, etc. but attackers soon discovered alternative ways to exploit memory errors.

Table 1: CERT Advisories (1999-2002)

Vulnerability	**Number**	**Percentage**
Buffer Overflow	49	44.95%
Double Free	2	1.83%
Format Strings	9	8.26%
Backdoor/Trojan Horses	8	7.34%
Others	41	37.62
Total	**109**	**100%**

As it is well known, diversity plays an essential role for the survivability of every biological species; recently this theory has also been applied to computer programs. Computer scientists started to apply different types of transformations to computer

programs such as Instruction Set Randomization (ISR), Absolute Address Randomization (AAR), Address Space Layout Randomization (ASLR), and several other program transformation techniques. These techniques either help in combating against memory corruption attacks or extenuate the chances of them being successful.

The recent release of Microsoft's latest PC operating system Windows Vista included an implementation of Address Randomization [6] which if analyzed is not up to the security requirements of current systems. Wehntrust [7] and Ozone [8] are other implementations available for address randomization. These products have one or more of the following negatives:

Randomization Incompleteness: Windows Vista's Address Randomization randomizes only the base addresses of the executables and the DLLs. Wehntrust also don't randomize some memory regions and for Ozone no other information is available except it randomizes the stack and the DLLs.

Randomization Deficiency: In Windows Vista Address Randomization is provided only over 256 possible values for the base address. This range of randomization hardly seems to be sufficient to counter a targeted attack. The attacker may use Brute-Force techniques to succeed and needs a mere 128 tries, on an average, before succeeding.

1.2 Motivation

Our initial work involved the study of system security related issues and we found that buffer overflow is a major threat to system and internet security. These attacks are not just a few kind but many kinds emerge as the time passes and break the security mechanism developed for the previous attack methods. As described above, many

techniques have evolved from the time the buffer overflow attacks came into light but no technique has been successful in preventing the majority of these attacks. Studying these attack methods and the security mechanism motivated us to design a system which can counter against the vast choices of these kinds of attacks. Our system not only provides the randomization at load time of the process but also at the runtime and with the application of ***cryptographic techniques*** and ***prioritization mechanisms***.

1.3 Objective

The goal of this thesis is to study various randomization techniques for process memory organization with the application of cryptographic techniques to make them more secure and to design a randomization technique which can work with existing architectures on top of the system kernel. The applicability of the design to the existing architectures & applications requires the design to be application & infrastructure transparent.

1.4 Proposed Work

In this thesis we propose an improved version of the idea of diversity which, besides being simpler, is able to handle a broader range of memory error exploits. In particular we make the following contributions:

1. We apply a cryptographic model for data transformation. The data when being written in memory is scrambled using some transformation function and when being used is unscrambled using the reverse transformation function.
2. We formulate a model which combats memory corruption attacks based on knowledge of absolute addresses as well as those which partially overwrite a memory address. The first attack type refers to all those exploitation techniques

an attacker may use to corrupt a particular memory address value with the objective of hijacking the process execution control flow.

1.5 Thesis Organization

The thesis is organized as follows:

Chapter 2 describes the various kinds of buffer overflow attacks and the techniques which have developed so far to counter these attacks & then conclude how these techniques are weak in preventing such attacks.

In **Chapter 3** we describe our system, i.e. PriCryp's Model and the various components of the design.

Then in **Chapter 4** we evaluate the design against various kinds of buffer overflow attacks, compare the model with other techniques and describe the limitations of the design.

In **Chapter 5**, we demonstrate our initial implementation work related to the PriCryp's model and the other techniques.

Finally we conclude in **Chapter 6** and describe our future work related to the work done in the thesis. At the last the publication made from the thesis are listed. These include the accepted paper and the communicated journals with some of the related details.

Chapter 2. Memory Exploitation & Preclusion

2.1 Memory ill-treatment Techniques

2.1.1 Introduction

A buffer overflow occurs in a program when the program stores more information in an array, the buffer, than the space reserved for it. This causes the areas adjacent to the buffer to be overwritten, corrupting the values previously stored there. Buffer overflows are always programming errors which are typically introduced into a program because the programmer failed to anticipate that the information copied into the buffer by the program may exceed its size. Unfortunately, as we shall soon see, buffer overflow programming errors are quite common because of certain widely used and dangerous C programming practices. Once buffer overflow vulnerability is present in a program, inadequate testing may not uncover it, so that the vulnerability may lurk in the program hidden, undiscovered and silent for years. This potentially opens up the program to be the target of a sudden attack which exploits the vulnerability to gain unauthorized access to a system. A buffer overflow may happen accidentally during the execution of a program. When this happens, however, it is very unlikely that it will lead to a security compromise of the system. Most often the clobbering of information in areas adjacent to the buffer will cause the program to crash or produce obviously incorrect results. In a buffer overflow attack, on the other hand, the objective of the attacker is to use the vulnerability to corrupt information in a carefully designed way in order to execute attack code previously planted by the attacker. If this succeeds, the attacker effectively hijacked the

control of the program. Once control is transferred to the attack code, it grants unauthorized access to the attacker.

Typically the attack code just spawns a shell, which allows the attacker to execute arbitrary commands on the system. When a new shell is spawned in a UNIX system, it inherits the access privileges of the process that spawned it. Consequently, if the attacked process containing the buffer overflow vulnerability runs with root privileges, the attacker will also get a root shell.

A buffer overflow attack may be local or remote. In a local attack the attacker already has access to the system and may be interested in escalating her access privilege. A remote attack is delivered through a network port, and may achieve simultaneously both gaining unauthorized access and maximum access privilege.

Summarizing, we see that a buffer overflow attack usually consists of three parts:

1. The planting of the attack code into the target program;
2. The actual copying into the buffer which overflows it and corrupts adjacent data structures;
3. The hijacking of control to execute the attack code;

2.1.1.1 Basic Example

In the following example, a program has defined two data items which are adjacent in memory: an 8-byte-long string buffer, A, and a two-byte integer, B. Initially, A contains nothing but zero bytes, and B contains the number 3. Characters are one byte wide.

A	A	A	A	A	A	A	A	B	B
0	0	0	0	0	0	0	0	0	3

Now, the program attempts to store the character string "excessive" in the A buffer, followed by a zero byte to mark the end of the string. By not checking the length of the string, it overwrites the value of B:

A	A	A	A	A	A	A	A	B	B
'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	'e'	0

Although the programmer did not intend to change B at all, B's value has now been replaced by a number formed from part of the character string. In this example, on a big-endian system that uses ASCII, "e" followed by a zero byte would become the number 25856. If B was the only other variable data item defined by the program, writing an even longer string that went past the end of B could cause an error such as a segmentation fault, terminating the process.

2.1.2 Exploiting Buffer Overruns

A buffer overrun is characterized as a *stack buffer overrun* or *heap buffer overrun* depending on what memory gets overrun. C and C++ compilers typically use the stack for local variables as well as parameters, frame pointers, and saved return addresses. Heaps, in this context, refer to any dynamic memory implementations such as the C standard library's malloc/free, C++'s new/delete, or the Microsoft Windows APIs HeapAlloc/HeapFree.

Published general-purpose exploits for buffer overruns typically involve two steps:

1. Change the program's flow of control. (Pure data exploits in which the buffer happens to be adjacent to a security-critical variable operate without changing the program's flow of control

2. Execute some code (potentially supplied by the attacker) that operates on some data (also potentially supplied by the attacker).

The term *payload* refers to the combination of code or data that the attacker supplies to achieve a particular goal (for example, propagating a worm). The attacker sometimes provides the payload as part of the operation that causes the buffer overrun, but this need not be the case. All that is required is that the payload be at a known or discoverable location in memory at the time that the unexpected control-flow transfer occurs.

The exploits can be broadly divided into two categories: those that exploit the call-stack and those that exploit the heap. Exploits on the stack include stack-smashing, Arc-Injection, function pointer clobbering, and exception handler hijacking. Heap-based exploits include attacks on function pointers, C++ *vtables*, executable sections and the malloc internal data-structure. First we will look at stack exploits.

2.1.2.1 Stack Based Exploits

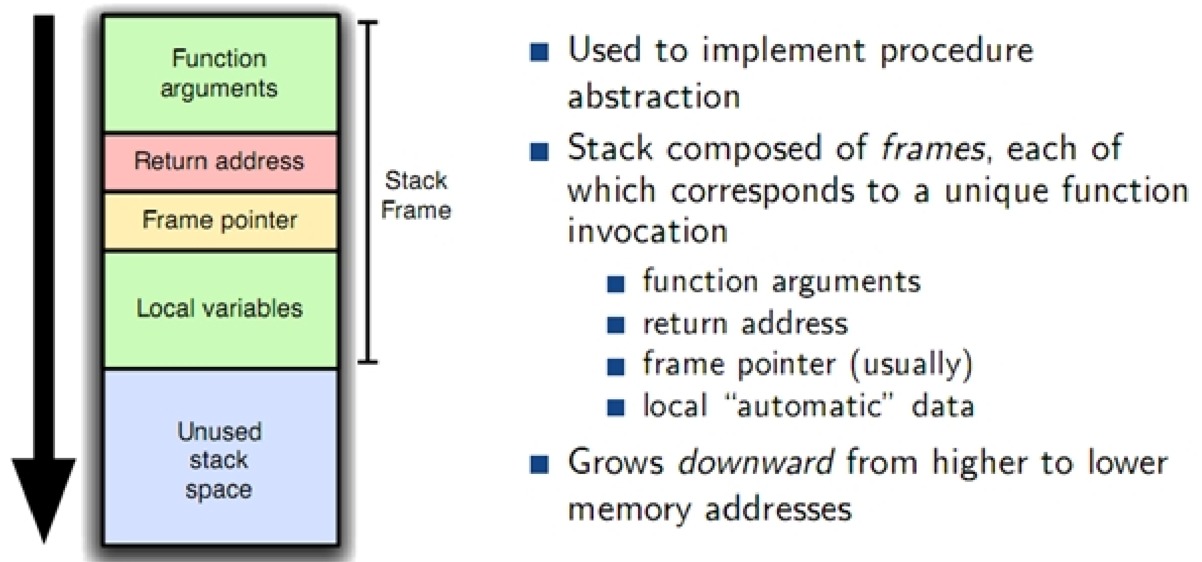

Figure 1: Structure of a Stack in a Process

2.1.2.1.1 Stack Smashing

One classification of buffer overflow attacks depends on where the buffer is allocated. If the buffer is a local variable of a function, the buffer resides on the run-time stack. This is by far the most prevalent form of buffer overflow attack.

When a function is called in a C program, before the execution jumps to the actual code of the called function, the activation record of the function must be pushed on the run-time stack. In a C program the activation record consists of the following fields:

1. Space allocated for each parameter of the function;
2. the return address;
3. the dynamic link;
4. Space allocated to each local variable of the function.

The address of the dynamic link field is to be considered as the base address of the activation record. The function must be able to access its parameters and local variables. This requires that during the execution of the function a register hold the base address of the activation record of the function, i.e. the address of the dynamic link field. Parameters are below this address on the stack, and local variables above. When the function returns, this register must be restored to its previous value, to point to the activation record of the calling function. To be able to do this, when the function is called the value of this register is saved in the dynamic link field. Thus the dynamic link field of each activation record points to the dynamic link field of the previous activation record on the stack, which in turn points to the dynamic link field of the previous activation record, and so on, all the way to the bottom of the stack. The first activation record on the stack is that of main (). This chain of pointers is called the dynamic chain.

In many C compilers the buffer grows towards the bottom of the stack. Thus if the buffer overflows and the overflow is long enough the return address will be corrupted, (as well as everything else in between, including the dynamic link.) If the return address is overwritten by the buffer overflow so as to point to the attack code, this will be executed when the function returns. Thus, in this type of attack, the return address on the stack is used to hijack the control of the program.

Overwriting the return address, as explained above, gives the attacker the means of hijacking the control of the program, but where should the attack code be stored? Most commonly it is stored in the buffer itself. Thus the payload string which is copied into the buffer will contain both the binary machine language attack code (shellcode) as well as the address of this code which will overwrite the return address. There are a few

difficulties that the attacker must overcome to carry out this plan. If the attacker has the source code of the attacked program it may be possible to determine exactly how big the buffer is and how far it is from the return address, determining how big the payload string must be. Also, the payload string cannot contain the null character since this would abort the copying of the payload into the buffer. Some copying routines of the C library use carriage returns and new lines as a delimiter instead, so these characters should also be similarly avoided in the payload string.

2.1.2.1.2 Arc Injection

As an alternative to supplying executable code, an attacker might simply be able to supply data that—when a program's existing code operates on it—will lead to the desired effect. One such example occurs if the attacker can supply a command line that the program under attack will use to spawn another process; this essentially allows arbitrary code execution. Arc injection exploits are an example of this data-oriented approach—indeed, the first such published exploit allowed the attacker to run an arbitrary program. The term "arc injection" refers to how these exploits operate: the exploit just inserts a new arc (control-flow transfer) into the program's control-flow graph, as opposed to code injection-exploits such as stack smashing, which also insert a new node.

2.1.2.1.3 Function Pointer Clobbering

Function-pointer clobbering is exactly what it sounds like: modifying a function pointer to point to attacker supplied code. When the program executes a call via the function pointer, the attacker's code is executed instead of the originally intended code. This can be an effective alternative to replacing the saved return value address in situations in

which a function pointer is a local variable (or a field in a complex data type such as a C/C++ struct or class).

2.1.2.1.4 Exception Handler Hijacking

Several variations of exploit techniques target the Microsoft Windows Structured Exception Handling (SEH) mechanism. When an exception (such as an access violation) is generated, Windows examines a linked list of exception handlers (typically registered by a program as it starts up) and invokes one (or more) of them via a function pointer stored in the list entry. Because the list entries are stored on the stack, it is possible to replace the exception-handler function pointer via buffer overflow (a standard example of function pointer clobbering), thus allowing an attacker to transfer control to an arbitrary location—typically a trampoline to code injected by the attacker. Versions of Windows starting with Windows Server 2003 perform some validity checking of the exception handlers that limit the feasibility of this straightforward attack. An alternative to clobbering an individual function pointer is to replace the field of the thread environment block (a per-thread data structure maintained by the Windows operating system) that points to the list of registered exception handlers. The attacker simply needs to "mock up" an apparently valid list entry as part of the payload and, using an arbitrary pointer write, modify the "first exception handler" field.

2.1.2.2 Heap Based Exploits

2.1.2.2.1 Heap Structure

Heap is a reserved address space region at least one page large from which the heap manager can dynamically allocate memory in smaller pieces. The heap manager is

represented by a set of function for memory allocation/freeing which are localized in two places: ntdll.dll and ntoskrnl.exe.

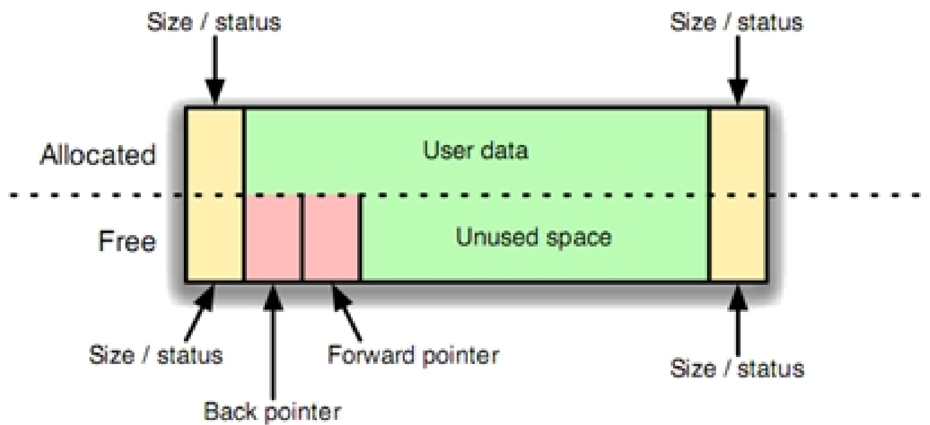

Figure 2: General Structure of a Heap

2.1.2.3 Heap Smashing

The key insight behind heap smashing is to exploit the implementation of the dynamic memory allocator by violating some assumed invariants. (Published source code for the dynamic memory allocator makes explanation easier, but is not needed in practice, so these techniques apply equally well on closed-source systems.) Many allocators, for example, keep headers for each heap block chained together in doubly linked lists of allocated and freed blocks, and update these during operations such as freeing a memory block. If there are three adjacent memory blocks *X*, *Y,* and *Z*, an overrun of a buffer in *X* that corrupts the pointers in *Y*'s header can thus lead to modification of an arbitrary memory location when *X*, *Y*, or *Z* is freed. In many cases, the attacker can also control the value being put into that location, thus accomplishing an arbitrary memory write, which leads to the exploitation possibilities. In practice, heap smashing is thus typically coupled with function-pointer clobbering. Three factors complicate heap-smashing exploits. Most

obviously, the attacker typically does not know the heap block's location ahead of time, and standard *trampolining* approaches are typically not effective. In many cases, it is somewhat difficult to predict when the heapFree operation will occur, which could mean that the payload is no longer available at the time that the call via the clobbered function pointer occurs. Finally, in some situations (especially with multithreaded programs), it is difficult to predict whether the next block has been allocated at the time the overrun occurs. Surprisingly enough, however, these are no longer significant roadblocks for exploitation of many heap buffer overruns. Attackers typically work around them by transferring the payload to an easy-to-find location as part of a separate operation (a technique first developed as a stack-smashing enhancement). There are typically enough such locations available that attackers can choose a location that will still be available at the time the call occurs. For cases where it is difficult to predict the next block, attackers can attempt to influence the size and order of heap operations to position the heap block of interest at a known location and to disambiguate the following blocks' behavior.

2.2 Thwarting Memory Corruption

Many techniques have been proposed till date to prevent and to detect memory corruption attacks, but no technique till date has been successful in countering all types of vulnerabilities. The technique proposed here is also not defending against all the vulnerabilities but this defends against most of them and more effectively.

The main proposed techniques for memory corruption related vulnerabilities are the following:

2.2.1 StackGuard

StackGuard defends against stack smashing [11] attacks as shown in figure 4. The stack smashing attacks usually overwrite the return address within the stack and perform the attack using *return-into-libc* [12][27] technique. StackGuard places a canary bit next to the return address in the stack and before jumping to the return address the canary bit is checked as shown in figure 4. If it's corrupt the process is considered to be compromised. But StackGuard is the compile time technique, i.e. the programs are compiled using this technique to make use of canary bit because of which its use is limited. This technique too addresses only specific types of attacks and cannot be used to defend against current day attacks such as those which modify code pointers in the static or code area. These types of attacks successfully bypass the canary bit protection and execute the shellcode.

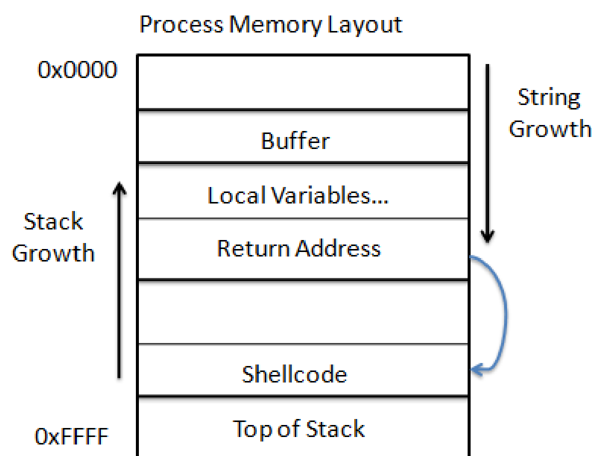

Figure 3: Stack Smashing Attack

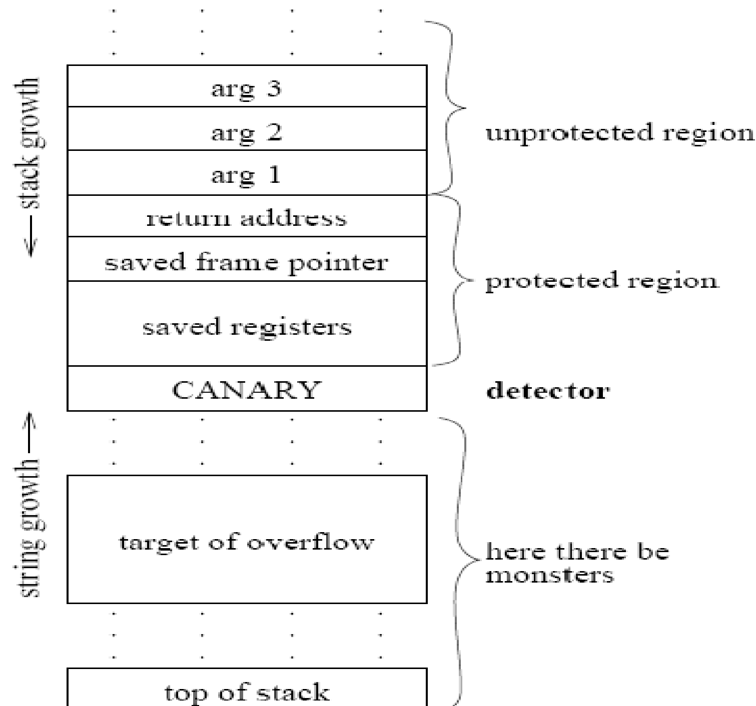

Figure 4: i386 Stack Layout with StackGuard

2.2.2 Code Obfuscation

The word "obfuscation" literally translates to "making something less clear and harder to understand". Code Obfuscation is a method to transform the program into a different internal architecture but keeping the semantics, i.e. the instructions will be transformed or even the gap between the instructions will be changed but the original and the transformed programs will work the same. This is the concept taken from the diversity in biological species. So one type of attack designed for one copy of the program may not work successfully on another copy of the program. But these techniques were usually employed at compile time or link time, i.e. they usually required source code of the program to transform them into a different design. Because of these disadvantages these

techniques were never used for the commodity software which is the biggest source of available vulnerabilities on the internet.

2.2.3 Absolute Address Randomization

This technique randomizes the absolute address of the various segments of a process [17][18]. At the process startup the very components of the process including the code segment, data segment are randomized within the process memory space. These blocks are randomized but the relative distance between these is kept the same. This type of technique can prevent several pointer corruption attacks because attacker cannot predict the object pointed by the corrupted pointer. Through this stack smashing attacks can be prevented. The problem with this technique, however, is it doesn't randomize the relative distances between segments. The *Relative Address* attacks which don't rely on data's absolute locations can defeat AAR techniques. Also AAR depends on the *secrecy* of *randomization key*. Since it's hard to keep it secret from local users AAR is basically limited to defend against the remote attacks.

2.2.4 Instruction Set Randomization

ISR creates randomized instruction set for each process at startup. Even the two images of the same program may have different internal architecture. This technique is an enhancement of the code obfuscation technique which was basically a compile time technique. The addresses of the instructions within the code segment are randomized before startup and thus the attacker cannot locate any desired instruction directly. With ISR in effect attacker fails to execute the injected code even if she has already corrupted the victim process's control flow. But ISR also suffers from the type of attacks which don't use injected code. The ISR cannot prevent the *return-into-libc* attacks which call a

library or system function rather than the injected code and pass the malicious arguments in this function to perform the attack tasks as shown in figure 5.

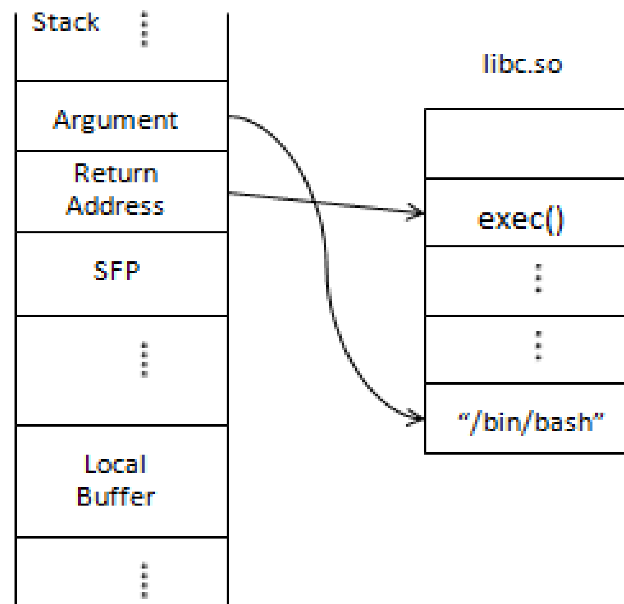

Figure 5: Return into libc Attack

2.2.5 NX/XD Bit

The **NX** bit, which stands for **N**o e**X**ecute, is a technology used in CPUs to segregate areas of memory for use by either storage of processor instructions (or code) or for storage of data, a feature normally only found in Harvard architecture processors. However, the NX bit is being increasingly used in conventional Von Neumann architecture processors, for security reasons.

Any section of memory designated with the NX attribute means that it's only to be used for storing data, so that processor instructions should not reside there, and cannot be executed if they do. The general technique, known as executable space protection, is used to prevent certain types of malicious software from taking over computers by inserting

their code into another program's data storage area and running their own code from within this section. Microsoft employed this technique in their PC operating system, Windows XP SP2, with the name of **DEP** [15] (**D**ata **E**xecution **P**revention). Windows XP SP2 used software based DEP to protect against memory corruption attacks on the processors which don't support NX/XD bit.

If an application attempts to run code from a protected page, the application receives an exception with the status code STATUS_ACCESS_VIOLATION. If your application must run code from a memory page, it must allocate and set the proper virtual memory protection attributes. The allocated memory must be marked PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, or PAGE_EXECUTE_WRITECOPY when allocating memory. Heap allocations made by calling the **malloc** and **HeapAlloc** functions are non-executable.

2.2.5.1 Hardware Enforced DEP

Hardware-enforced DEP marks all memory locations in a process as non-executable unless the location explicitly contains executable code. There is a class of attacks that attempt to insert and execute code from non-executable memory locations. DEP helps prevent these attacks by intercepting them and raising an exception.

Hardware-enforced DEP relies on processor hardware to mark memory with an attribute that indicates that code should not be executed from that memory. DEP functions on a per-virtual memory page basis, usually changing a bit in the page table entry (PTE) to mark the memory page.

The actual hardware implementation of DEP and marking of the virtual memory page varies by processor architecture. However, processors that support hardware-enforced DEP are capable of raising an exception when code is executed from a page marked with the appropriate attribute set.

Both Advanced Micro Devices (AMD) and Intel Corporation have defined and shipped Windows-compatible architectures that are compatible with DEP.

Beginning with Windows XP Service Pack 2, the 32-bit version of Windows utilizes the no-execute page-protection (NX) processor feature as defined by AMD or the Execute Disable bit feature as defined by Intel. In order to use these processor features, the processor must be running in Physical Address Extension (PAE) mode. The 64-bit versions of Windows XP uses the NX processor feature on 64-bit extensions and certain values of the access rights page table entry (PTE) field on IPF processors.

2.2.5.2 Software Enforced DEP

An additional set of data execution prevention security checks have been added to Windows XP SP2. These checks, known as software-enforced DEP, are designed to mitigate exploits of exception handling mechanisms in Windows. Software-enforced DEP runs on any processor which is capable of running Windows XP SP2. By default, software-enforced DEP only protects limited system binaries, regardless of the hardware-enforced DEP capabilities of the processor.

But these both, software enforced DEP and hardware enforced DEP can be bypassed and the attack can be performed as shown in the recent studies [16].

2.2.6 Address Space Layout Randomization [17]

ASLR randomizes the memory layout of an executing process including the DLLs, heaps, stack and the gap between the data and the code segments. This technique can be seen as an enhancement to the previous Absolute Address Randomization techniques because in this scheme all of the sections of a process are randomized and the relative distance between them as well. So in this scheme even if the attacker succeeds in injecting a malicious code, *shellcode*, becomes hard to locate and execute. This way it prevents the attacker hijacking the process's control flow. However ASLR is also not *unbreakable*. It is easy to guess the shellcode location in this scheme too using the techniques like *Code Spraying* and *AddressSpraying* [18]. Code Spraying is a technique where attacker "sprays" the shellcode repetitively over a large writable user-level memory area and thus leaves only a little range for attacker to guess the location of shellcode. Also the limited range of randomization as used in the recent release of Microsoft's PC operating system, Windows Vista, is vulnerable to Brute Force attacks and the attacker can succeed in 128 tries on an average because of the total 256 available locations of randomization.

2.2.7 DAWSON [19]]

According to this proposed technique, a DLL is injected into every process before the start of its execution. This DLL hooks Windows API functions relating to memory allocation and randomizes the base address of all the regions. Further some of the randomizations is provided with a custom designed loader which randomizes the memory allocated prior to the injection of described DLL. This technique doesn't randomize the relative distances between objects and also there is no security for the data overwrites.

This technique cannot defend against relative distance attacks and the ones caused by data overwrite.

2.2.8 ASLP (Address Space Layout Permutation) [20]

Address Space Layout Permutation, as proposed, makes use of both ISR and ASLR to prevent such kind of attacks. However this technique randomizes all the instructions without prioritizing the instructions which are usually used to accomplish the attacker's target. Further it doesn't provide any mechanism to secure data from overwrite and the misuse of these overwrites. In our view the randomization of all the instructions is inefficient because not all the instructions are security critical. Typically the injected shellcode makes use of system calls to harm victim machine and to replicate, thus from our perspective the technique is a little expensive.

2.2.9 RISE (Randomized Instruction Set Emulation) [27]

RISE uses a machine emulator to transform a program into a diverse program at runtime. The transformed program has a different, secret instruction set. The machine emulator in this technique produces automatically diversified instruction sets. In RISE each byte of original program code is scrambled using pseudorandom numbers seeded with a unique random key unique to each program in execution. Since this technique scrambles each instruction, it is real slow. And again there is no other technique other than scrambling which defends the system and if the secret key is compromised the system becomes vulnerable. This technique thus cannot defend against local attacks.

2.3 Conclusion

The techniques developed so far have some or more weaknesses which certainly cause the risks to the system security. We identified some weaknesses which need to be overcome to better secure the system from memory corruption attacks.

1. The use of source code transformation is not at all efficient & also not possible for commodity software which is the major source of vulnerabilities on the web.
2. The NX/XD Bit prevents write & execute at the same time but recent studies have shown that this technique can also be bypassed.
3. ISR techniques randomize the instructions but keep the data intact. The attackers can overwrite the data & can hijack the process control flow by overwriting the data & passing wrong arguments to the system functions, i.e. return into libc.
4. AAR techniques randomize the absolute address of the various components of the process including code segment, data, stacks, heaps, DLLs. But the relative distances are kept same. The attacker can succeed if the attack doesn't depend on the absolute address and can be done according to the relative address.
5. ASLR also employs only one kind of randomization, i.e. randomization at load time. But the techniques like code spraying and address spraying can break the system.
6. ASLP is a better technique than the ones before but it also takes no care of data and also no weight is given to the security critical instructions. Also, no runtime randomization.

7. RISE transforms the process instruction set into a totally different instruction set but this technique seems to be infeasible and only one secret key is used to create this random set.

Chapter 3. PriCryp Model

3.1 Design Goals

Observing the above techniques we noticed that these have many positives but many negatives too. Many defend against many types of attacks but no technique defends against most of the attacks. In this thesis we advocate and demonstrate that by mixing up ISR and ASLR and enhancing the ASLP technique with some cryptography, priority assignment and data encryption we can offer much more robust protection technique than each of them can provide individually.

In this thesis we suggest the use of a set of components which individually enable the system to overcome the weaknesses previously described and together make the system robust enough to defend against most of the memory corruption attacks.

1. The randomization techniques today randomize the components of the process like stacks, heaps, code segments, data, & DLLs. We suggest a fine grained randomization inside the code segments. This can be accomplished with the help of a binary rewriting tool which analyses the process code at load time & decomposes it into several segments & then randomize these segments.
2. To defend against de-randomization attacks we suggest runtime randomization according to some kind of prioritized mechanism. What we do in this thesis is assign the priority to the code blocks according to the security critical instructions within them. We assume system calls to be the security critical instructions.

3. Again to defend against control flow hijacking & the return-into-libc attacks we propose the application of cryptographic algorithms to transform some of the important instructions to some unidentifiable form.
4. To prevent the data overwrites and the memory corruptions attacks through them we suggest data scrambling using some cryptographic algorithms.

To integrate these ideas into a complete system and to design a mechanism to defend against memory corruption attacks we propose the PriCryp design in the next section, which has many components each doing some important function in system and together enabling the system to defend against the so called memory error exploits. The suggested components of the system and the sequence of their execution are shown in figures 6 and 8 respectively.

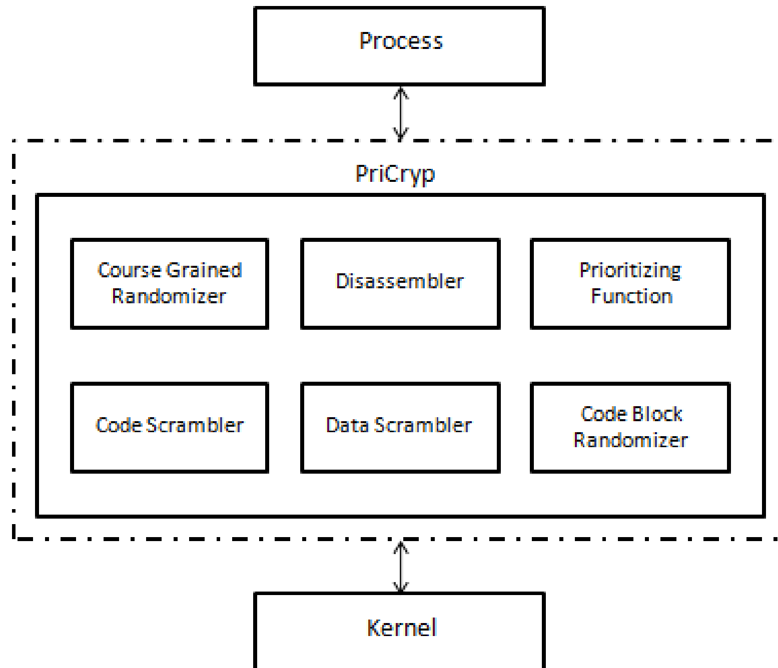

Figure 6: PriCryp Components and the interface with the Process and the Kernel

3.2 PriCryp Design

What we propose here is the randomization on a per code block basis, which can be a specific set of instructions identified by PriCryp's runtime service interface. Besides this we also employ some cryptographic transformations to transform static data in data section and a fraction of the important instructions within program's code. PriCryp performs randomization at both load-time and runtime. At load time it performs randomization through a thin transparent virtualization layer which is basically a system service and consists of a disassembler to identify the code fragments within the executable and de-randomizes them at runtime through the mapping table stored in the memory. PriCryp would run as a system specific service and would be completely transparent to the process, enabling each process to run in a secure environment.

PriCryp has many components for randomizations & various cryptographic transformations. In this section we describe our course grained randomizer, fine grained randomizers and various cryptographic transformations.

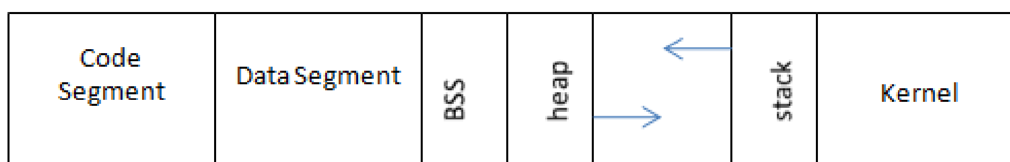

Figure 7: Normal Process Memory Layout

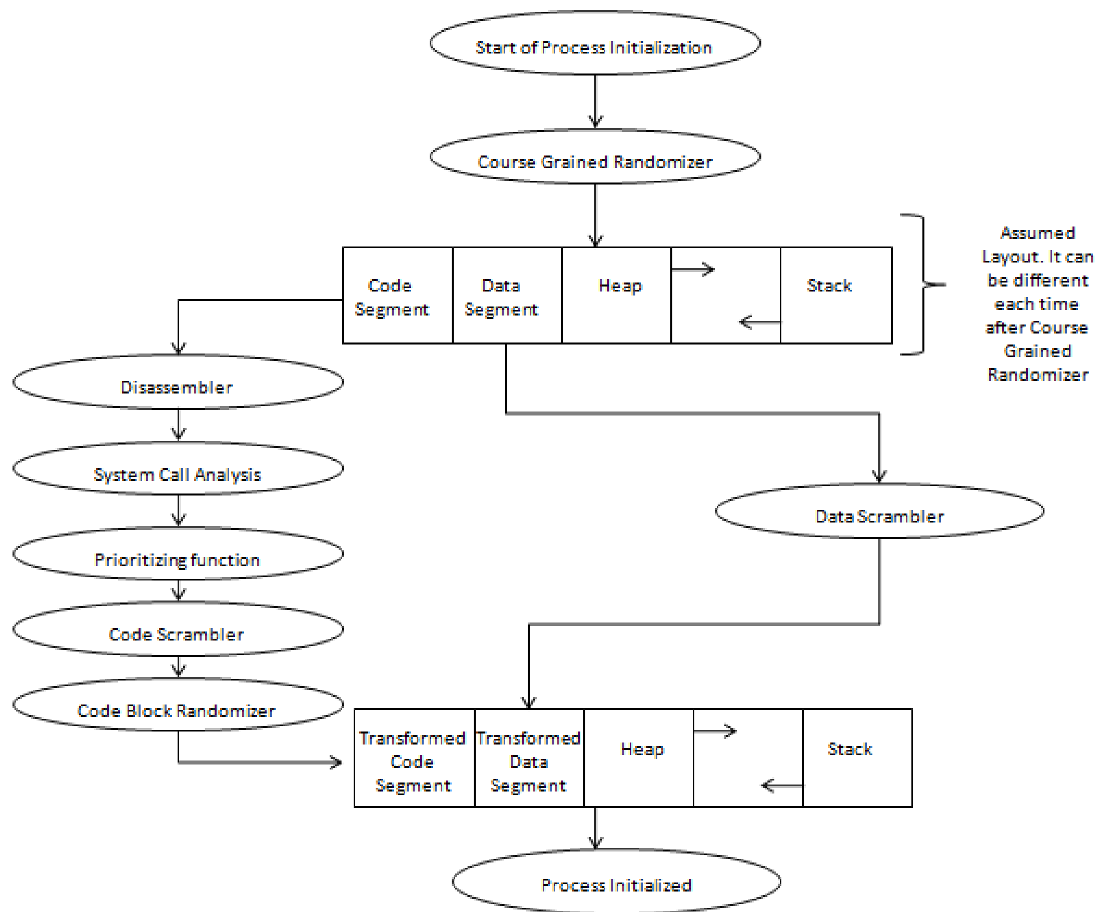

Figure 8: PriCryp's program flow design

3.2.1 Course Grained Randomizer

This component is the implementation of Address Space Randomization in the kernel. The components of the process including code segment, data segment, stacks, heaps, and DLLs are randomized. The components are randomized in such ways that the relative distance between them is also unpredictable. This makes the system more powerful to defend against memory corruption attacks even if the attacker succeeds in guessing the absolute location of one segment. The relative distance randomization is achieved using padding. The padding incurs wastage of memory space but it becomes reasonable when it

comes to defense against memory corruption attacks. The unpredictability of future randomizations causes the future attack attempts to fail. And because of relative address randomization the brute force attacks also become more difficult if not impossible. The outcome of the course grained randomizer can be as shown in figure 9, extreme left, as opposed to the normal process layout of figure 7.

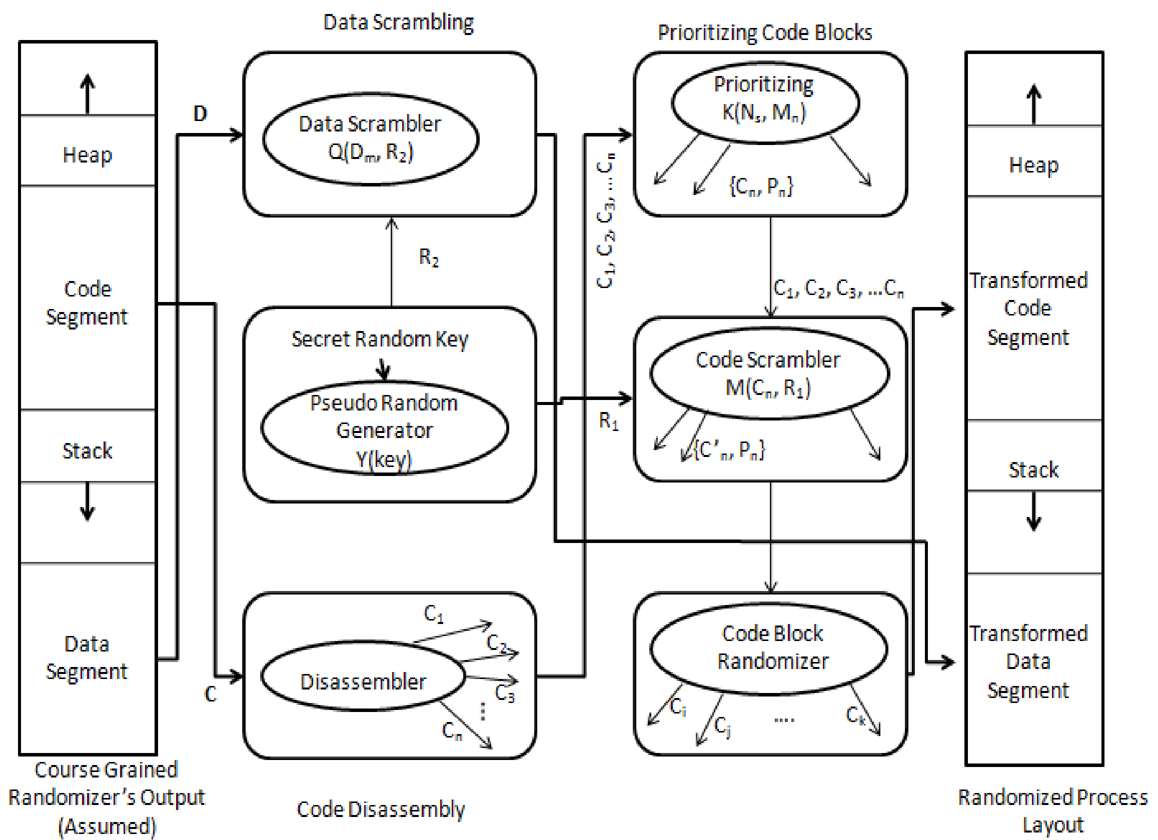

Figure 9: PriCryp Design Internals

3.2.2 Disassembler

Disassembler is the part of our binary rewriting tool. At the time of creation of the process the PriCryp takes control of the process by intercepting *sys_execv* call to the kernel to perform the randomizations & transformations. The disassembler works on the

code segment only and performs a quick analysis of the code segment to identify the instruction blocks of usually a single high level function as shown in figure 10. These instruction blocks are marked by the start address and the end address and are given proper numbers, e.g. C_1, C_2…, C_n etc. A conceptual example is shown in figure 11. The figure shows two instruction blocks identified by the disassembler.

$$C = (C_1, C_2, C_3\ldots, C_n) \tag{1}$$

```
00401308  |. 6A 00          PUSH 0                                  ; /Style = MB_OK|MB_APPLMODAL
0040130A  |. 68 4C004200    PUSH OFFSET DEP.??_C@_05EFDD@ERROR?$AA@  ; |Title = "ERROR"
0040130F  |. 68 2C004200    PUSH OFFSET DEP.??_C@_0BJ@NMND@Something>; |Text = "Something is screwed up!"
00401314  |. 6A 00          PUSH 0                                  ; |hOwner = NULL
00401316  |. FF15 E8624200  CALL DWORD PTR DS:[<&USER32.MessageBoxA>>; \MessageBoxA
0040131C  |. 3BF4           CMP ESI,ESP
0040131E  |. E8 0D040000    CALL DEP.__chkesp
00401323  |. 33C0           XOR EAX,EAX
00401325  |. EB 72          JMP SHORT DEP.00401399
==========================================================================================================
00401399  |> 5F             POP EDI
0040139A  |. 5E             POP ESI
0040139B  |. 5B             POP EBX
0040139C  |. 81C4 90000000  ADD ESP,90
004013A2  |. 3BEC           CMP EBP,ESP
004013A4  |. E8 87030000    CALL DEP.__chkesp
004013A9  |. 8BE5           MOV ESP,EBP
004013AB  |. 5D             POP EBP
004013AC  \. C2 1000        RETN 10
```

Figure 10: Two Code Blocks Identified Using Disassembler

```
00401308  |. 6A 00          PUSH 0                                  ; /Style = MB_OK|MB_APPLMODAL
0040130A  |. 68 4C004200    PUSH OFFSET DEP.??_C@_05EFDD@ERROR?$AA@  ; |Title = "ERROR"
0040130F  |. 68 2C004200    PUSH OFFSET DEP.??_C@_0BJ@NMND@Something>; |Text = "Something is screwed up!"
00401314  |. 6A 00          PUSH 0                                  ; |hOwner = NULL
00401316  |. FF15 E8624200  CALL DWORD PTR DS:[<&USER32.MessageBoxA>>; \MessageBoxA
0040131C  |. 3BF4           CMP ESI,ESP
0040131E  |. E8 0D040000    CALL DEP.__chkesp
00401323  |. 33C0           XOR EAX,EAX
00401325  |. EB 72          JMP SHORT DEP.0040721D
==========================================================================================================
0040721D  |> 5F             POP EDI
0040721E  |. 5E             POP ESI
0040721F  |. 5B             POP EBX
00407220  |. 81C4 90000000  ADD ESP,90
00407226  |. 3BEC           CMP EBP,ESP
00407228  |. E8 87030000    CALL DEP.__chkesp
0040722D  |. 8BE5           MOV ESP,EBP
0040722F  |. 5D             POP EBP
00407230  \. C2 1000        RETN 10
```

Figure 11: Randomized Code Blocks of figure 10

3.2.3 Prioritizing

On inspecting a number of memory corruption attacks we observe that the system calls are almost always used by the malicious codes to perform their malicious tasks such as replication, data corruption, etc. Here in this thesis we assign priorities to the code blocks so that the randomizations can be done effectively according to the priority. The code block with least number of system calls will be assigned the lowest priority and the block with the highest number of system calls will be assigned the highest priority. A block's priority basically determines the frequency of runtime randomizations of that block.

$$P_i = K(N_s, M_n) \qquad (2)$$

Where,

K = Prioritizing Function,

N_s = Number of system calls within i^{th} Block,

M_n = Number of blocks having same number of system calls.

The prioritizing process is done after the disassembling and the input to the prioritizing are the code blocks identified by disassembler as shown in the figure 9.

3.2.4 Code Scrambler

In this component of PriCryp we generate a diverse instruction set for some of the instructions in the program's code segment. As suggested in RISE, generating an automatic diverse instruction set for each instance of the process helps defending against some of code injection attacks. But that scheme is inefficient in a way that all the instructions are transformed. Here in this component we suggest transforming only

security critical functions; we assume system call wrapper functions & other application binary interface (ABI) functions as security critical, which are vital for an attacker to successfully perform the malicious tasks on the victim. We keep other instructions unchanged and scramble the security critical instruction by a function having two inputs: a pseudo random number and the security critical instruction. We pass a secret random key to the pseudo random generator to generate a key to scramble the instruction.

$$R_1 = Y\ (secret\ random\ key) \quad (3)$$

$$C_i' = M\ (C_i, R_1) \quad (4)$$

$$C' = (C_1', C_2', C_3' \ldots, C_n') \quad (5)$$

Where,

R_1 = Pseudo Random Number,

Y = Pseudo Random Generator Function,

M = the scrambling function,

C_i = i^{th} Code Block.

C_i' = Scrambled C_i block.

The scrambling function here can be any reversible cryptographic algorithm like DES, IDEA, etc. The use of cryptographic algorithm makes the scrambling more powerful and robust. And the independence of choosing the algorithms in PriCryp makes it more favorable to be deployed.

The address of the scrambled instruction relative to the base address of the code block it belongs is stored in the PriCryp's virtualization layer's memory. The virtualization layer stores the data on a per process basis.

During execution whenever there is a call to the scrambled instruction, the instruction is unscrambled using the reverse of the algorithm employed in scrambling function and the instruction can be executed successfully.

3.2.5 Code Block Randomizer / Fine Grained Randomizer

3.2.5.1 Load Time Randomizer

This code block randomizer is nothing but the implementation of ASLR for fine grained randomizations within the code segment. The code blocks identified as C_1, C_2, C_3, etc. are randomized using the code block randomizer as shown in the figure 9. These randomizations are done without considering the priorities of the code blocks. The outcome of these randomizations is a transformed code segment as shown in the figure 9. This transformed code segment has the same functionality as the original code segment but a different internal architecture.

3.2.5.2 Runtime Randomizer

The runtime randomizer randomizes the code blocks identified in section 3.2.2 according to the priority level assigned to them in section 3.3.3. The frequency of randomization of code blocks depends on the priority, i.e. the number of security critical instructions within them. Since the blocks having no security critical instructions are assumed to be having no harm, they are not at all randomized during runtime.

$$\mathbf{F_i = Z\ (P_i)} \qquad (9)$$

Where,

F_i = Frequency of randomization of ith code block,

Z = Randomization Function,

P_i = Priority of ith code block.

3.2.6 Data Scrambler

The data segment D consists of many static data objects D1, D2, D3, etc. In current systems these data objects are stored in memory as they are. In these systems the attacker sometimes succeeds in overwriting the data with her malicious shellcode and then jump to it to execute it and perform the malicious task. What we propose here is to transform the data into some unidentifiable form before writing to memory at load time and revert it back to the original form before every use. In this scheme even if the attacker succeeds in overwriting the data during runtime and jump to it to execute, the runtime mechanism of PriCryp would first apply the reverse transformation algorithm on the data before making it usable. By this reverse transformation the attacker's shellcode would probably convert into useless binary strings and the execution of these would certainly crash the program.

$$D = (D_1, D_2, D_3 \ldots, D_m) \quad (6)$$

$$D_i' = Q\ (D_i, R_2) \quad (7)$$

$$D' = (D_1', D_2', D_3' \ldots, D_m') \quad (8)$$

Where,

D_i = i^{th} data object in Data Section.

D_i’ = Scrambled form of D_i,

Q = Any Cryptographic Scrambling Function,

R_2 = Secret Randomization Key for data transformation.

3.3 Design Strengths

1. PriCryp randomizes all the sections of the process memory as done in the previous techniques like ASLR, gaining all of the advantages of those techniques, & also overcomes the weaknesses of them by further security mechanisms like cryptographic application, runtime randomization, and prioritization of the security critical instruction related code blocks.
2. PriCryp provides a robust, probabilistic technique with prioritization of the security critical instructions. This enables the system to withstand a huge variety of memory corruption attacks.
3. The application of cryptographic techniques in the system to encrypt the instructions and the data variables prevents the overwrites of data and instructions to work correctly.
4. The prioritized runtime randomization adds additional security to the system to counter the de-randomization attacks which are the most recent in the generation of buffer overflow attacks.

Chapter 4. Evaluation & Limitations

4.1 Evaluation

We evaluate the design of our proposed technique, PriCryp, from the following aspects:

4.1.1 Unintentional Memory Corruption

These memory corruption exploits occur due to the flaws in programs. These would certainly result in the crash of the program due to the randomizations & the cryptographic techniques and would certainly cause no harm to the system.

4.1.2 Traditional Attacks

We consider the stack smashing, heap overflow, stack overflow, & return into libc attacks to be the traditional attacks due to their age of usage.

Stack smashing attacks make use of shellcode to execute harmful code within the stack. First of all the attacker needs to have the knowledge of memory allocation within process to overwrite the data with shellcode and then execute & then to successfully execute the shellcode she needs to be aware of all the cryptographic keys used to scramble the security critical instructions and the data variables. The randomizations within PriCryp makes it almost impossible for the attacker to find out the memory map of the process & even if she succeeds in this, the shellcode would be unscrambled into a probably unusable form before execution because the system think of it to be the data which was scrambled at the time of write to the memory.

Almost the same happens with the heap overflow attacks & they would result in the crash of the process rather than even start of execution of the attack code.

The return into libc attack so far are considered to be the most dangerous attacks in the category of memory corruption attacks but the use of cryptographic techniques to scramble data before writing it to memory & unscrambling it before using make this attack unsuccessful until the attacker knows the secret cryptographic keys used to scramble the data & instructions. And since in this design we make use of two different secret keys, we assume it to be difficult for the attacker to guess or find the both of them.

4.1.3 Chained Return-into-Libc Attacks

These attacks are the modern form of the traditional return-into-libc attacks. The prior techniques are not sufficient to defend against *chained return-into-lib(c) attacks*, each of which calls a sequence of system library functions in order. These attacks need the location of all the target functions in order to exploit the attack. And due to the multi-level randomizations in PriCryp it becomes difficult to trace the location mapping of the functions. Thus the chained return-into-libc attacks are almost impossible to succeed under PriCryp.

4.1.4 Brute Force Attacks

The brute force attacks are also the least possible in PriCryp due to the many rounds of randomizations and the application of cryptographic transformations. In case of ASLR brute force attack had the chances of success for 128 tries because of only 256 locations of randomization. In PriCryp we can achieve the randomization up to 2^{32} for 32 bit architecture by creating the code blocks to the basic building blocks, i.e. one code block for every instruction. This way the PriCryp handsomely defends against brute force attacks and such an attack would certainly result in the crash of the program in the early phase of the attack.

4.1.5 De-Randomization Attacks

Recently De-Randomization attacks came into the existence to counter the ASR defense. These attacks are able to de-randomize the components of the process and find out the exact location of the needed component. But these are also limited to the depth of randomization. These can de-randomize the effects of ASR but not the effects of PriCryp due to the depth of randomization in the design. In PriCryp we have randomizations after randomizations, i.e. after randomizations at the course level, we randomize at the fine level too. Further the prioritized runtime randomizations just make these attacks the benign attacks and cause the program to crash early.

4.2 PriCryp in Contrast with Others

In this section we compare our proposed mechanism with the existing techniques against various kinds of buffer overflow related attacks and technique features as shown in the table 2 on the next page.

4.3 Limitations

As with other techniques, PriCryp is also not the single sufficient for all of the attacks. PriCryp also has many limitations as described below:

Since PriCryp has many randomization components and uses cryptographic techniques to secure data and security critical instructions, it is slower than the previous techniques. The load time overhead is very low but the runtime randomization causes more overhead.

The PriCryp model is designed such that it makes use of secret keys to encrypt and decrypt the data and security critical instructions. On leakage of these secret keys the

system cannot prevent the memory corruption attacks crafted carefully with de-randomization mechanisms.

Table 2: Comparison between PriCryp and Others

Techniques Attacks & Features	StackGuard	ISR	NX/XD	ASLP	RISE	PriCryp
Unintentional	Y	Y	Y	Y	Y	Y
Stack Smashing	Y	N	N	Y	Y	Y
Chained Return into Libc	N	N	N	Y	Y	Y
Return into Libc	N	N	N	N	Y	Y
Code Injection	N	Y	N	N	Y	Y
De-Randomization	N	N	N	N	N	Y
Address Spraying	N	N	N	N	N	Y

Chapter 5. Implementation Details

Our initial work in this project work included the study of various kinds of buffer overflow attacks and the related defense techniques as described in chapter 2 and how to bypass these techniques. In this chapter we demonstrate how we can bypass the NX/XD mechanism to successfully execute our shellcode and perform the desired task. This implementation is an initial work towards the PriCryp's implementation for the complete understanding of the security breaches.

5.1 Tools Used

1. Visual C++ 6.0
2. OllyDBG
3. Dependency Walker
4. Metasploit Project

5.1.1 Visual C++ 6.0

In this project I have used Microsoft Visual C++ 6.0 for coding because the project needed windows console applications. Again since DEP in windows doesn't check applications written in .NET or cygwin based compilers it became mandatory.

5.1.2 OllyDBG

OllyDBG is a very powerful debugger with good user interface and disassembly features. This enabled to find out the addresses of the functions in running mode and to control the execution of the applications step through step while providing much information regarding current assembly instruction and its address in the executable.

5.1.3 Dependency Walker

Dependency Walker provides, with good user interface, all the linked libraries that are used in the application. All the functions of used libraries in applications are shown and the system base addresses of library functions can be observed.

5.1.4 Metasploit Project

Metasploit [13] Project is an online database of various exploits found in various Operating Systems. This provides creating the required shellcode for execution in Windows & Linux like operating systems and for use in many languages like C/C++, Ruby, & Perl.

5.2 Bypassing DEP

It should be mentioned that upon startup, every application in its memory space, maps certain dynamically linked libraries which are needed for the proper functioning of the application. There are two libraries that are always present in the application's address space. These libraries are ntdll.dll and kernel32.dll. We mention these two dynamically linked libraries because they are mapped in a memory region that is marked as executable. By using these dynamically linked libraries we can execute functions that are stored in them. This type of attack is called *Return to libc*.

In order to gain control over the program we need to overflow the stack in such a way that we will overwrite the saved return address with the address of the WinExec functions. The WinExec function is part of the dynamically linked library ntdll.dll. After the address of WinExec we need a memory address which will be used for WinExec to return to. At the end of the exploited string we will put the arguments of WinExec(),

which are going to be executed after we take control over the program. Figure 12 illustrates our attack string.

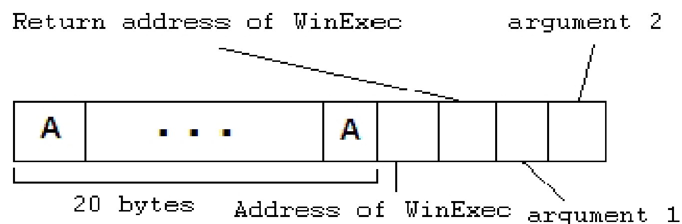

Figure 12: The return to libc attack string

The program that constructs the exploit string is the following:

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
main (int argc, char **argv) {
        char buf[512];
        memset (buf,'A',sizeof(buf));
        *(long*)&buf[20] = 0x77c293c7;
         *(long*)&buf[24] = 0x7c81caa2;
        *(long*)&buf[28] = getenv("envx");
        buf[32] = 0x00;
        printf ("%s\n",buf);
}
```

Our exploit fills the buffer as shown in Figure 12. As it can be seen from the exploit, the first argument is stored in an environment variable and the second argument is null. The first argument contains a command prompt command (dir, del). We can see that Data Execution Prevention did not stop the attack. The answer why it did not stop the attack

can be found in the dynamically linked library ntdll.dll. WinExec and all functions that can be found in ntdll.dll were stored in a memory region that was executable, making our attack successful. The attack that uses WinExec is limited to execution of one command at a time, which is an issue if the attacker wants full control of the system. During initialization of the memory, Data Execution Prevention uses *VirtualAlloc* and *VirtualProtect* to manage the non-executable feature. By using *VirtualProtect* we could try to mark the memory region where our code is stored as executable and then jump to our code. The second idea that arises with *return into lib* is to copy our malicious code to a static memory location that is marked as executable and call the malicious code from there. Both of the mentioned attack vectors can be used successfully to take control of applications/services.

Every process at creation time is granted with a default heap, which is 1MB large (by default) and grows automatically as need arise. The default heap is used not only by the win32 apps, but also by many runtime library functions which need temporary memory blocks. A process may create and destroy additional private heaps by calling *HeapCreate ()*/*HeapDestroy ()*. Use of the private heaps` memories is established by calling *HeapAlloc ()* and *HeapFree ()*.

Memory in heaps is allocated by chunks called 'allocation units' or 'indexes' which are 8-byte large. Therefore, allocation sizes have a natural 8-byte granularity. For example if an application needs a 24-byte block the number of allocation units it gets 3 allocation units. In order to manage memory for every block a special header is created, which also has a size divisible by 8 (fig. 13, 14). Therefore a true memory allocation size is a total of the

requested memory size, rounded up towards a nearest value divisible by 8 and the size of the header.

Size		Previous Size	
Segment Index	Flags	Unused	Tag Index

Figure 13: Busy Block Header

Size		Previous Size	
Segment Index	Flags	Unused	Tag Index
Flink			
Blink			

Figure 14: Free Block Header

Where:

Size - memory block size (real block size with header / 8);

Previous Size - previous block size (real block size with header / 8);

Segment Index - segment index in which the memory block resides;

Flags - flags:

- 0x01 - HEAP_ENTRY_BUSY

- 0x02 - HEAP_ENTRY_EXTRA_PRESENT

- 0x04 - HEAP_ENTRY_FILL_PATTERN

- 0x08 - HEAP_ENTRY_VIRTUAL_ALLOC

- 0x10 - HEAP_ENTRY_LAST_ENTRY

- 0x20 - HEAP_ENTRY_SETTABLE_FLAG1

- 0x40 - HEAP_ENTRY_SETTABLE_FLAG2

- 0x80 - HEAP_ENTRY_SETTABLE_FLAG3

Unused - amount of free bytes (amount of additional bytes);

Tag Index - tag index;

Flink - pointer to the next free block;

Blink - pointer to the previous free block.

The specification of the allocation size in allocation units is important for the free block list management of Heaps. Those free block lists are sorted by size and the information about them is stored in an array of 128 doubly-linked-lists inside the heap header (fig. 15, 16, 17). Free blocks in the size diapason from 2 to 127 units are stored in lists corresponding to their size (index). For example, all free blocks with the size of 24 units are stored in a list with index 24, i.e. in Freelist[24]. The list with index 1 (Freelist[1]) is unused, because blocks of 8 bytes can`t exist and the list with index 0 is used to store blocks larger than 127 allocation units (bigger than 1016 bytes).

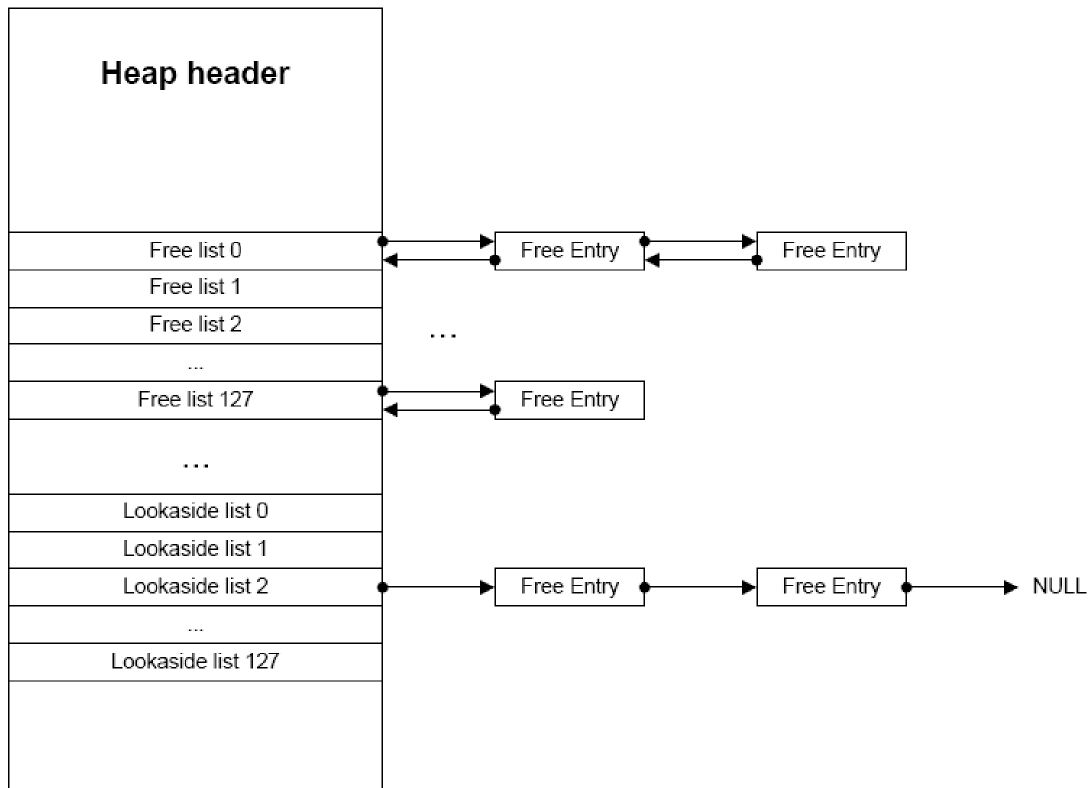

Figure 15: Lookaside Lists for Heap Management

If, during the heap allocation, the HEAP_NO_SERIALIZE flag was unset but the HEAP_GROWABLE flag was set (which is actually the default), then in order to speed up allocation of the small blocks (under 1016 bytes) 128 additional singly-linked lookaside lists (fig. 15, 16, 17) are created in the heap. Initially lookaside lists are empty and grow only as the memory is freed. In this case during allocation or freeing these lookaside lists are checked for suitable blocks before the Freelists.

The heap allocation routines automatically tune the amount of the free blocks to store in the lookaside lists, depending on the allocation frequency for certain block sizes. The more often memory of certain size is allocated -- the more can be stored in the respective lists, and vice versa -- underused lists are trimmed and the pages are freed to the system.

Because the main goal of the heap is to store small memory blocks this scheme results in relatively quick memory allocation/freeing.

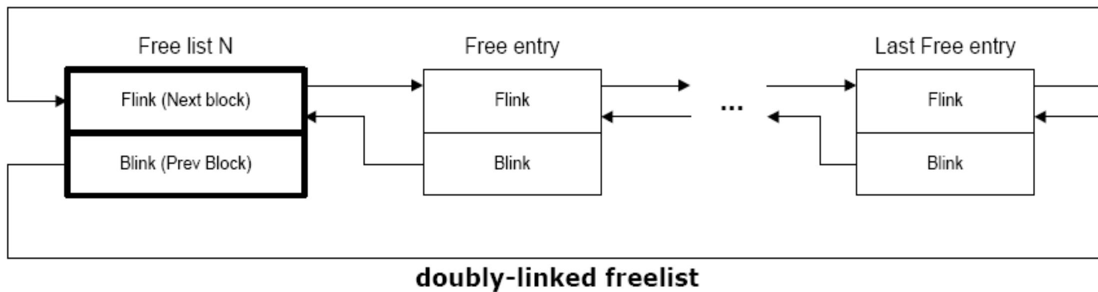

Figure 16: Freelist Entries of free heap blocks

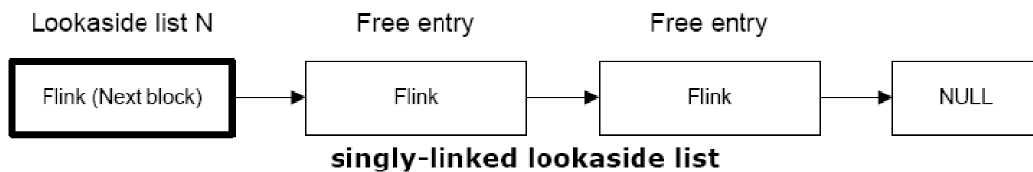

Figure 17: Lookaside List of heap blocks

The heap overflow exploitation scenario usually proceeds on like this:

If during the buffer overflow the neighboring block exists, and is free, then the Flink and Blink pointers are replaced (Fig. 18).

At the precise moment of the removal of this free block from the doubly-linked freelist a write to an arbitrary memory location happens:

```
mov dword ptr [ecx],eax

mov dword ptr [eax+4],ecx
```

EAX - Flink

ECX - Blink

For example, the Blink pointer could be replaced by the unhandled exception filter address (UEF -- UnhandledExceptionFilter), and Flink, accordingly, by the address of the instruction which will transfer there execution to the shellcode.

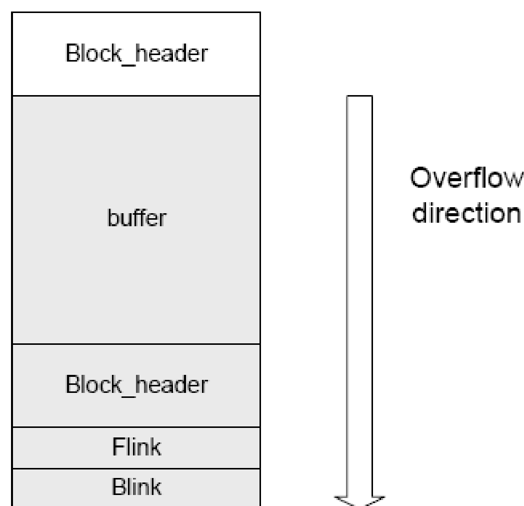

Figure 18: Direction of Heap Overflow

In Windows XP SP2 the allocation algorithm was changed -- now before the removal of a free block from the freelist, a pointer sanity check is performed with regard to the previous and next block addresses (safe unlinking, fig. 19.):

```
1. Free_entry2 -> Flink -> Blink == Free_entry2 -> Blink
   -> Flink
2. Free_entry2 -> Blink -> Flink == Free_entry2
```

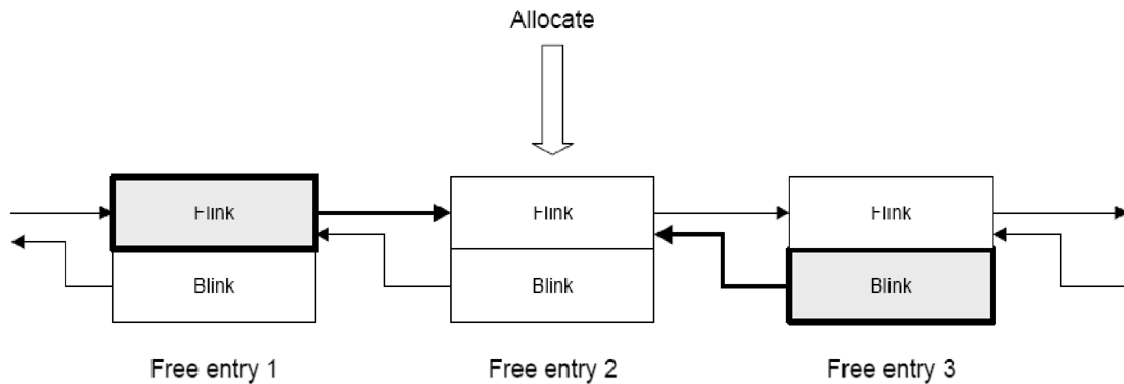

Figure 19: Safe Unlink Operation

Then that block gets deleted from the list. The memory header block was changed, besides other things (fig. 20.). A new one-byte-large 'cookie' field was introduced, which holds a unique pre-computed token -- undoubtedly designed to ensure header consistency. This value is calculated from the header address and a pseudorandom number generated during the heap creation:

(&Block_header >> 3) xor (&(Heap_header + 0x04))

The consistency of this token is checked only during the allocation of a free memory block and only after its deletion from the free list.

<table>
<tr><td colspan="2">Size</td><td colspan="2">Previous Size</td></tr>
<tr><td>Cookie</td><td>Flags</td><td>Unused</td><td>Segment Index</td></tr>
<tr><td colspan="4">Flink</td></tr>
<tr><td colspan="4">Blink</td></tr>
</table>

Figure 20: Free Block Header in Windows XP SP2 (Cookie Added)

If at least one of these checks fails the heap is considered destroyed and an exception follows.

The first weak spot -- the fact that the cookie gets checked at all only during free block allocation and hence there are no checks upon block freeing. However in this situation there is nothing you can do except changing the block size and place it into an arbitrary freelist.

And the second weak spot – the manipulation of the lookaside lists doesn`t assume any header sanity checking, there isn`t even a simple cookie check there, which, theoretically, results in possibility to overwrite up to 1016 bytes in an arbitrary memory location.

The exploitation scenario could proceed as follows:

If, during the overflow the coincidental memory block is free and is residing in the lookaside list, then it becomes possible to replace the *Flink* pointer with an arbitrary value.

Then, if the memory allocation of this block happens, the replaced *Flink* pointer will be copied into the header of the lookaside list and during the next allocation *HeapAlloc ()* will return this fake pointer.

The prerequisite for successful exploitation is existence of a free block in lookaside list which neighbors with the buffer we overflow.

The effect of a successful attack:

1. Arbitrary memory region write access (smaller or equal to 1016 bytes).
2. Arbitrary code execution.

3. DEP bypass.

5.3 Code

The Code consists of the following C++ files.

1. DEP.cpp
2. DEP1.c
3. DEP2.c

5.3.1 DEP.cpp

This file is the main GUI code in Visual C++ for executing the other both programs which bypass the data execution prevention.

5.3.2 DEP1.c

This file contains the code to bypass the software enforced DEP. It does so by making use of a shellcode crafted to execute the windows calculator and is manipulated to contain the address of system function from the dynamically allocated library msvcrt.dll. This shellcode is then copied on to an overflowed heap block so that the system function is executed. Since the system function resides on the executable pages our shellcode executes successfully and the calculator is executed. Every time the calculator is closed the system function returns to its own address because of overwrite so it executes again and again indefinitely.

5.3.3 DEP2.c

This file contains the code to bypass the hardware enforced DEP. It does so by making use of buffer overruns and lookaside list overwrites. By intelligently crafted lookaside list overwrite and buffer overruns we put the fake return address in the stack and copy our

shellcode to be executed and the address of the system function from dynamic library msvcrt.dll into heap. This way the system function executes without making windows aware of data execution prevention has been bypassed.

5.4 Screenshots

5.4.1 Bypassing Software Enforced DEP

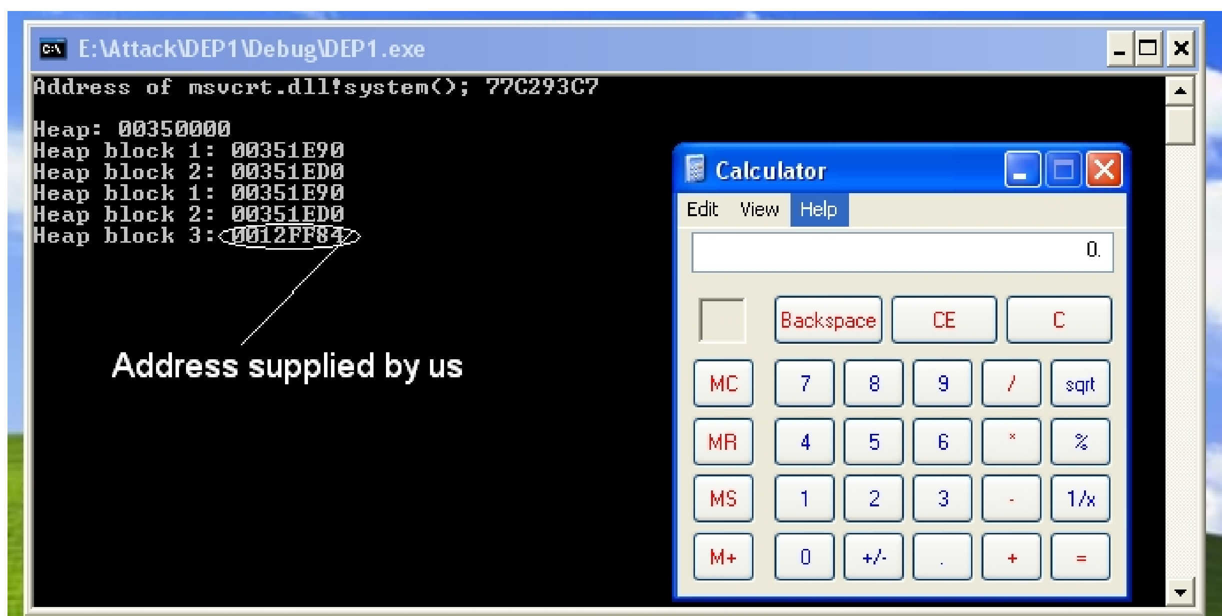

Figure 21: Shellcode execution bypassing software enforced DEP

As you can see the calculator executed as a result of the execution of our shellcode from within the system function taken from msvcrt.dll file which we dynamically loaded within our program and thus succeed in executing data from non-executable pages, i.e. bypassing software enforced data execution prevention. The shellcode, i.e. calculator executes again and again we close it until we close our console application.

5.4.2 Bypassing Hardware Enforced DEP

As can be seen from figure 21, the 3rd heap block is assigned the address which we supplied during the overflow and since we have copied our shellcode with address of system function here, our shellcode is executed from the data pages which are in fact non-executable and thus windows hardware enforced data execution prevention is bypassed and we succeed in executing the required shellcode.

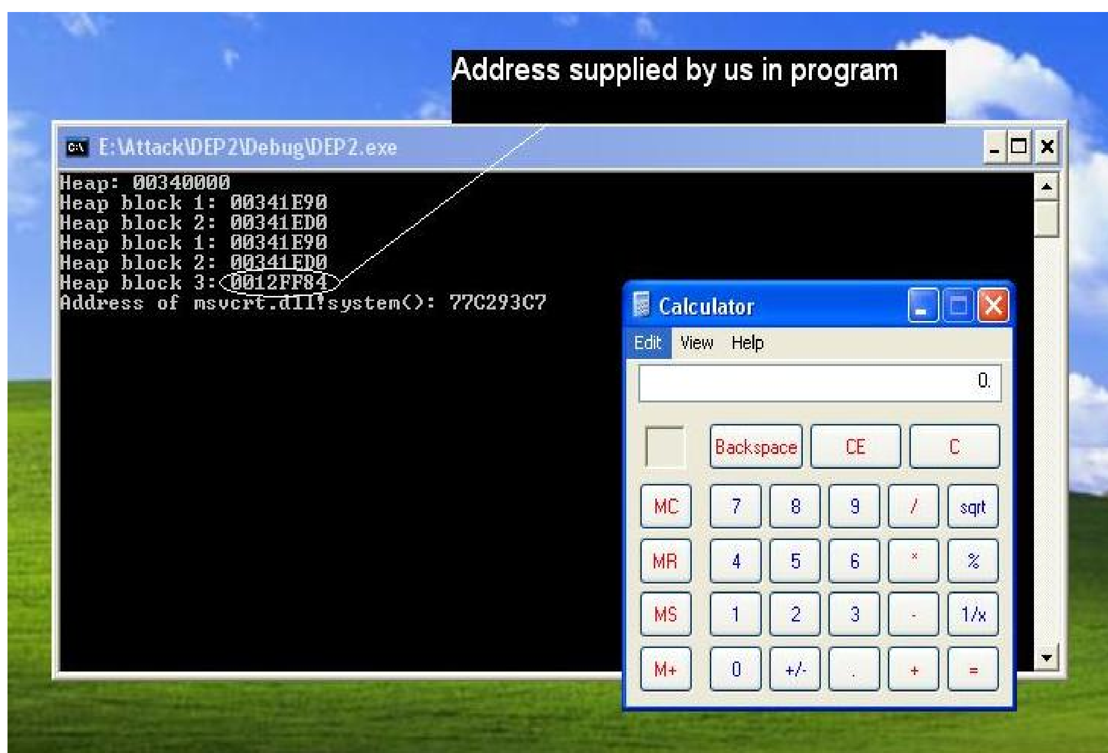

Figure 21: Shellcode execution bypassing hardware enforced DEP.

As you can see from the screenshot of the execution in figure 22, on closing the windows calculator which was executed as a result of hardware enforced DEP bypass, the windows detects that execution is going on from non-executable pages and so a windows pops out to warn and to close the application. But all this happens long after we did what we

wanted, i.e. executing our shellcode from the data pages and thus bypassing hardware enforced data execution prevention.

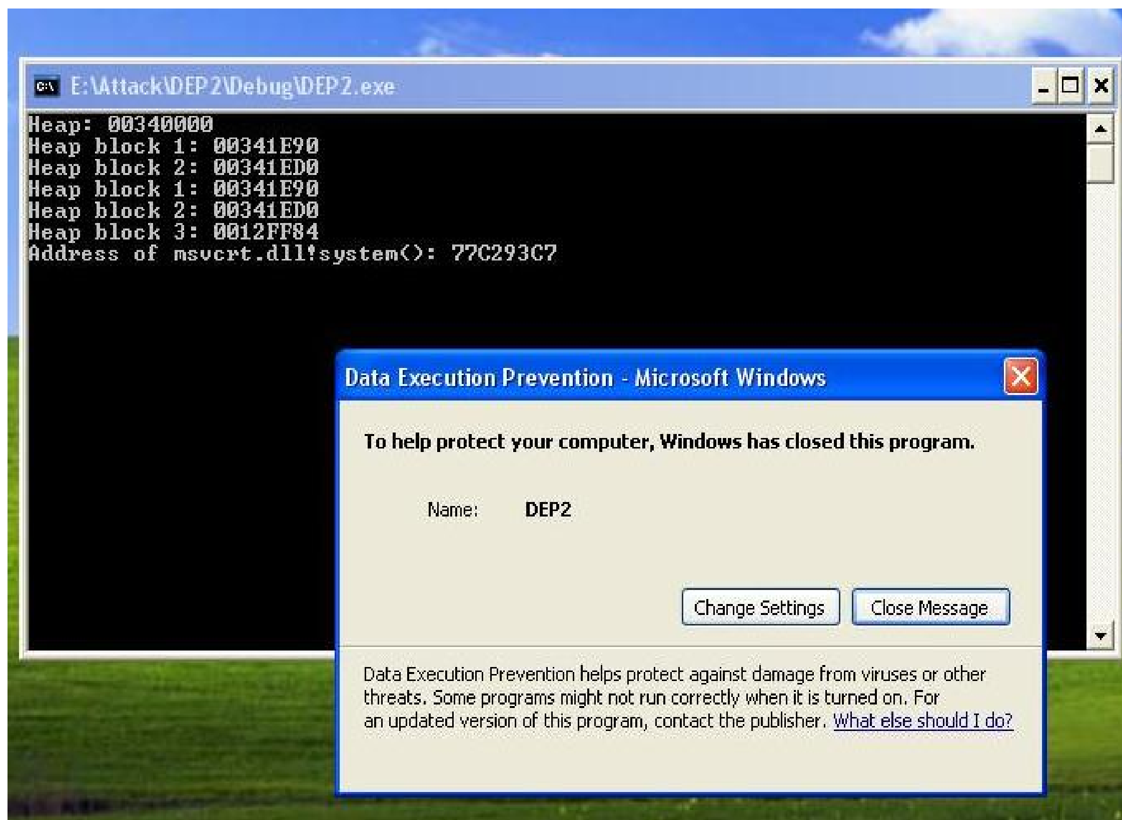

Figure 22: Popped Up Warning Message on Closing Windows Calculator

Chapter 6. Conclusion & Future Work

In this thesis we presented the theoretical aspects of the PriCryp design which helps defending against today's major source of vulnerabilities, i.e. the memory corruption. The major contributions of this thesis are:

- PriCryp provides probabilistic defense against memory corruption attacks by prioritizing the code blocks having security critical instructions. Since system calls or other application binary interface are almost always used by attackers to perform malicious tasks, it becomes difficult for the attacker to succeed in the presence of probabilistic runtime randomization.
- PriCryp employs cryptographic techniques to protect the system from attacks resulting from malicious data overwrites.
- PriCryp transforms the security critical instructions into unintelligible form to protect the instructions from being used harmfully by the attacker.
- PriCryp provides very fine control over the granularity of the randomization which can reach the basic building block (one instruction per code block) if desired. This provision provides favors for easy use and deployment in the variations of systems.
- PriCryp randomizes all the components of the process including stacks, heaps, code segments, data segments and DLLs. The relative distance between the segments is also randomized using padding.

- The startup performance overhead of PriCryp is very low because it does not randomize all the instructions, rather randomizes only code blocks having group of instructions.
- The performance overhead of PriCryp during runtime of the process is slightly higher, however, due to runtime randomization of the code blocks. But the compound overhead becomes reasonable when it provides probabilistic proactive defense mechanism against memory corruption attacks.

We believe that PriCryp has significant potential to extenuate the wide threat of memory corruption attacks. By randomizing the memory space holding the program the core vulnerability that memory corruption attacks use is addressed- namely the predictability of control information and critical data. Further the transformation of data variables and security critical instructions provides the system more strength against modern memory corruption attacks.

Our future work includes the implementation of the complete PriCryp design in Linux system & enhancing the application of cryptographic techniques in ASR to improve the design of PriCryp.

Chapter 7. Publications from the Thesis

7.1 Accepted Paper

Title: HighRAND: High Priority Instruction Block Randomizer

Abstract: *A majority of security exploits today are memory error exploits. Address Space Randomization is a broadening and promising method of preventing a vast range of memory corruption attacks. ASR shifts critical memory regions at process initialization time which causes an otherwise successful malicious attack to crash like a trivial attack. Insufficient randomness as provided in Microsoft's recent PC operating system Windows Vista (256 Locations), is also not likely to significantly slow down self replicating worms. On the other hand other existing techniques require source code modifications, which is an inefficient approach for commodity software. Here we propose High Priority Instruction Block Randomizer (HighRAND), which introduces randomness over randomness with minimal performance overhead. HighRAND is a prophylactic security technology that increases system security by increasing the diversity of attack targets. HighRAND runs as an Operating System Service in Microsoft Windows or a Kernel Module in a Linux System or others, which Randomizes the procedures in the executable's code segment. This tool disassembles the program to be executed and fragments the program in several parts giving the priority to the fragments having system calls and then randomizes the fragments over the code segment assigned by the Address Randomization powered kernel.*

Authors: Rahul K. Agrawal, Daya Gupta

Conference: Second International Conference on Information Processing

Website: http://www.icjip.org

Dates: 8-10 August, 2008

Venue: Bangalore, India

PC Members: R L Kashyap, Purdue University, USA

Dharma P Aggarwal, University of Cincinnati, USA

L M Patnaik, Indian Institute of Science, India

David Kahaner, AIIP, Japan

7.2 Communicated Journal

Title: PriCryp: Prioritized Runtime Cryptographic Randomization for Memory Corruption Extenuation

Authors: Rahul K. Agrawal, Dr. Daya Gupta

Journal: International Journal of Information Security, Springer Verlag

Communication Date: 27/06/2008

Chapter 8. References

[1] Sasser Worms: *http://www.microsoft.com/security/incident/sasser.asp*, *May 2004.*

[2] MSBlaster Worms. CERT Advisories CA-2003-20 W32/Blaster Worms *http://www.cert.org/advisories/ca-2003-20.html*, August 2003.

[3] Code Red Worms. CAIDA Analysis of Code Red Worms. *http://www.caida.org/analysis/security/code-red/*, 2001.

[4] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. The spread of the sapphire/slammer worm, 2003. Available from URL *http://www.cs.berkeley.edu/nweaver/sapphire/*

[5] CERT Advisories: *http://www.cert.org*

[6] Michael Howard. ASLR features in Windows Vista. Published on World-Wide Web at URL *http://blogs.msdn.com/michael_howard/archive/2006/05/26/608315.aspx*, 2006.

[7] WehnTrust, Published on World Wide Web at URL *http://www.wehnus.com/products.pl*, 2006.

[8] Eugune Tsyrklevich. Ozone. Published on World Wide Web at URL *http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-tsyrklevich.pdf*, 2005.

[9] Bobby D. Birrer, Richard A. Raines, Rusty O. Baldawin, Barry E. Mullins: Program Fragmentation as a Metamorphic Software Protection. In Third International Symposium on Information Assurance & Security.

[10] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer overflow attacks. In *Proceedings of the 7th USENIX Security Conference,* 1998.

[11] Jonathan Pincus, Brandon Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. Published on World-Wide Web at URL http://www.computer.org/security. IEEE Security & Privacy, 2004.

[12] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection attacks with instruction-set randomization. In *ACM Computer and Communications Security (CCS),* 2003.

[13] Metasploit Project http://www.metasploit.com

[14] Intel: Execute Disable Bit to defend against Buffer Overflow attacks. http://www.intel.com/business/bss/infrastructure/security/xdbit.htm

[15] Microsoft Corporation. Data Execution Prevention. www.microsoft.com/technet/prodtechnol/windowsserver2003/library/BookofSP1/b0de1052-4101-44c3-a294-4da1bd1ef227.mspx

[16] Nenad Stojanovski, Marjan Gusev, Danilo Gligoroski, Svein J. Knapskog: Bypassing Data Execution Prevention in Windows XP SP2, IEEE 2007. http://www.ieeexplore.ieee.org

[17] PaX Team: PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt

[18] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium,* 2003.

[19] Lixin Li, James E. Just, R. Sekar. Address-Space Randomization for Windows Systems. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06).*

[20] ChongKyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation: Towards Fine-Grained Randomization of

Commodity Software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06).*

[21] Manish Prasad and Tzi cker Chiueh. A Binary Rewriting Defense against Stack based Buffer Overflow Attacks. In *USENIX Annual Technical Conference,* 2003.

[22] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting pointers from Buffer Overflow Vulnerabilities. In *USENIX Security Symposium,* 2003.

[23] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *ACM Symposium of Principles of Programming Languages (POPL),* 2002.

[24] Nagendra Modadugu, Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, and Dan Boneh. On the effectiveness of address-space randomization. In V. Atluri, B. Pfitzman, and P. McDaniel, editors. *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004,* October 2004.

[25] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An efficient approach to combat a broad range of memory error exploits.

[26] Arash Baratloo, Timothy Tsai, and Navjot Singh. LibSafe: Protecting Critical Elements of Stacks

http://www.securityfocus.com/librar/2267

http://www.bell-labs.com/orgg/11356/libsafe.html

[27] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Darko Stefanovic, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. *ACM Transactions on Information and System Security,* 2005.