Software Security, DV2546
Lab 2-Buffer Overflow

Kaavya Rekanar
940521-7184
kare15@bth.se

Siva Venkata Prasad Patta
931221-7137
sipa15@bth.se

Introduction- The Buffer Overflow

Buffer overflow is a vulnerability in software systems affecting the data integrity in recent times.
A buffer overflow occurs when a program or process tries to store more data [3] in a buffer than it was initially intended to hold. Since buffers, which are temporary data storage area, are actually created to contain a finite amount of data, the extra information-which has to go somewhere-can overflow into adjacent buffers [3], corrupting or overwriting the valid data held in them. Although it may occur accidentally through program error, buffer overflow is an increasingly common type of security attack.
In buffer overflow attacks, the extra data may contain codes [3] to trigger certain actions, whose effect could be sending new instructions to the attacked computer that could [3], for example, damage the user's files, change data, or disclose confidential information.
This report contains a detailed account of buffer overflow vulnerability and it's exploitation using the source code given. The methods, tools and techniques used to do so have also been discussed in the report. The knowledge earned in this process is clearly described in the Reflexion section.

Task 1: Ennumeration of Common Vulnerabilities and Common Weaknesses

1.1 Purpose of the websites

1.1.1 Common Vulnerabilities and Exposures list

- CVE- Common Vulnerabilities and Exposures is a glossary of common names (identifiers) for information security vulnerabilities [1].
- The common identifiers in the list provide a standard to evaluate the coverage of security services and tools, which will help users in determining the most appropriate tools according to their needs [1].
- Common Vulnerabilities and Exposures provides a good coverage, easier inter-operability and enhanced security to the products and services, which are compatible with it [1].

1.1.2 Common Weakness Enumeration

- CWE- Common Weakness Enumeration is a catalogue of software weaknesses created to serve as a common language for describing software security loopholes in architecture, design or code [2].
- CWE serves as a standard measurement scale for software security tools to identify and mitigate weaknesses in architecture, design or code [2].
- CWE provides a common standard for identification of weaknesses, diminution and prevention efforts [2].

1.2 Difference between the websites

S.No.	Common Vulnerabilities and Exposure List (CVE)	Common Weakness and Enumeration (CWE)
1.	CVE deals with mistakes in softwares, which are usually exploited by hackers for their own advantage-known as software vulnerabilities [1].	CWE deals with exploitable software vulnerabilities, which are caused due to common software weaknesses which occur in software's architecture, design, code or implementation [2].
2.	CVE provides a standard for evaluating the coverage security tools and services through the CVE identifiers in the list [2]	CWE provides a standard for weakness identification, mitigation and prevention efforts [2].
3.	CVE compatible products need to satisfy the 4 criteria: CVE searchable, CVE output, mapping and documentation [1].	CWE compatible products need to satisfy 2 extra criterions apart from the four mandatory requirements in order to be more effective [1].

Table 1: Differences between CVE and CWE

1.3 Relation between the websites

Both Common vulnerabilities and exposures (CVE) and Common Weakness Ennumeration (CWE) have been developed by MITRE Corp.
MITRE's CVE team worked on the issue of grouping software weaknesses into different categories when they launched the list.
They developed a fundamental classification and categorization of vulnerabilities, attacks, faults to help them define common software weaknesses [2]. These groupings were elementary and needed improvement in order to identify the functionality offered in the security industry. Hence, the CWE list has been created with the help of CVE list [2].

Task 2: Disable stack protection of the compiler

Vulnerability and Method

The operating systems currently in use have built-in protection against all attacks by the hackers. They are being made in a secure fashion.
Disabling the security mechanism has to be known to disable the security mechanism. This task requires us to disable stack protection and other security features of the compiler by executing a few flags according to the selected operating system. The OS we have selected is Ubuntu 15.10.
The flags used are:

1. -fno-stack-protector :
   Emits extra code to check for buffer overflows[3], such as stack smashing attacks[4]. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call alloca, and functions with buffers larger than 8 bytes[4]. The guards are initialized when a function is entered and then checked when the function exits[4]. If a guard check fails, an error message is printed and the program exits[4].

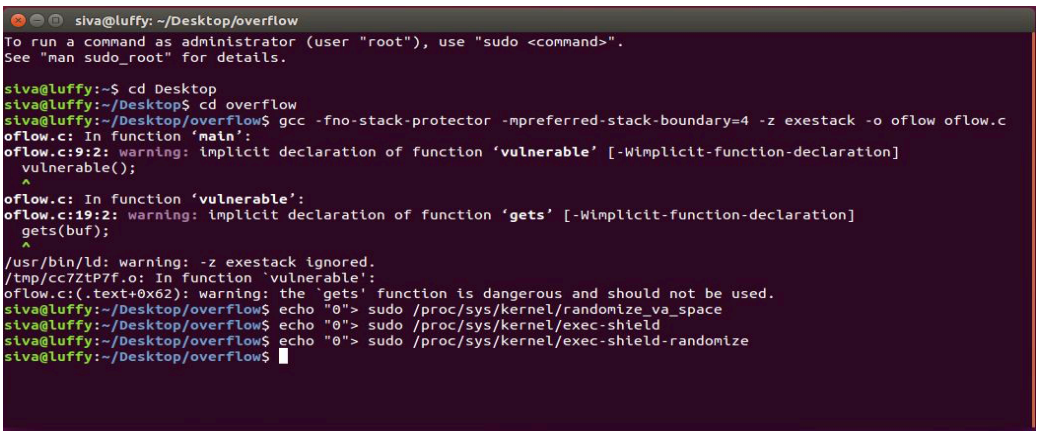
- -fstack-protector-all
  works similar to -fstack-protector except that all functions are protected.

2. –mpreferred-stack-boundary :
   This boundary flag limits the size of the stack, but it is unsafe to use as there is a possibility that this flag might crash the code that is being executed [3].
   This flag affects generated code in your binary[3]. By default, GCC will arrange things so that every function, immediately upon entry, has its stack pointer aligned on 16-byte boundary (this may be important if you have local variables, and enable sse2 instructions)[3].
   If you change the default to e.g. -mpreferred-stack-boundary=2, then GCC will align stack pointer on 4-byte boundary. This will reduce stack requirements of your routines, but will crash if your code (or code you call) *does* use sse2, so is generally not safe.

3. –z execstack :
   The executable stacks are the regions, which are most vulnerable to exploitation for any hacker. Hence, the usage of this flag results in enabling the stack and disables stack protection thus giving a chance to raise loopholes in the security mechanism [3].

4. ASLR :
   Address Space Layout Randomization is a technique, which is used as a protection from buffer overflow attacks.
   ASLR mitigation adds a significant component in exploit development, but we realized that sometimes a single module without ASLR loaded in a program can be enough to compromise all the benefits at once.
   For this reason recent versions of most popular Microsoft programs were natively developed to enforce ASLR automatically for every module loaded into the process space. In fact, *Internet Explorer 10/11 and Microsoft Office 2013* are designed to run with full benefits of this mitigation and they *enforce ASLR randomization natively without any additional setting on Win7 and above*, even for those DLLs not originally compiled with /DYNAMICBASE flag. So, customers using these programs have already a good native protection and they need to take care only of other programs potentially targeted by exploits not using ASLR.

   Making use of these flags, we executed respective commands in terminal in Ubuntu 15.0 Operating System.



These commands have been implemented on oflow.c source code as that would prove helpful for the next task. Stack protection is diabled when this code is compiled, which eventually results in exploiting the overflow security mechanism is the operating system.
gcc and gdb are the tools used in completion of this task,i.e., disabling stack protection. GCC (GNU Compiler Collection) works as a compiler for C programs in Linux. Gdb is a standard debugger in GNU operating system.

Task 3: Execution of function 'notcalled'

Vulnerability and Method:

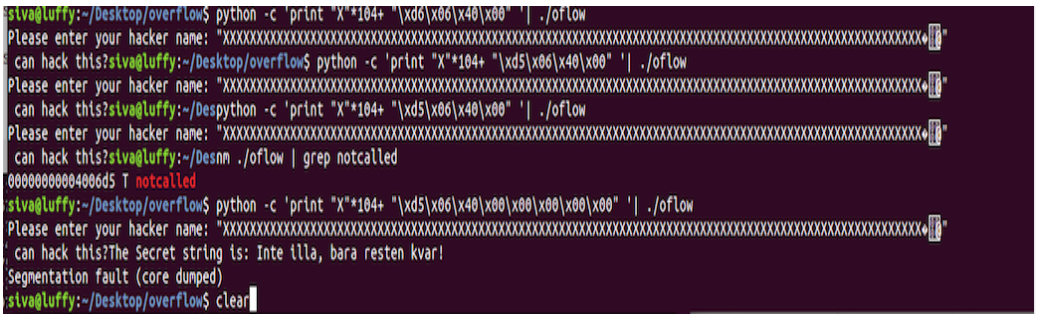
An unsafe function gets() is being used in oflow.c which made the code vulnerable to buffer overflow attack.
This vulnerability can be exploited and made to suit our needs as required for the task. The work we planned to do for this is, overflow the buffer and then plan an execution of the function notcalled().
To do so, the address of the function is needed. As given in the material, the command **nm <filename> | grep <function>** can be used as **nm <oflow.c> | grep <notcalled>** which resulted in the address 0x4006d5. This address can be used to cause a buffer overflow attack.

```
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from oflow...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x40065a
(gdb) break notcalled
Breakpoint 2 at 0x4006d9
(gdb) run
Starting program: /home/siva/Desktop/overflow/oflow

Breakpoint 1, 0x000000000040065a in main ()
(gdb) continue
Continuing.
Please enter your hacker name: kakaka
"kakaka"
 can hack this?[Inferior 1 (process 22029) exited normally]
(gdb) x/201 main
0x400656 <main>:        -443987883      552371016       1223458185      1222669705
0x400666 <main+16>:     16270791        -956301312      62533   12058624
0x400676 <main+32>:     -402653184      7       184     1438894336
0x400686 <vulnerable+1>:        1223002440      -1083118461     4196376 184
0x400696 <vulnerable+17>:       -25892864       -1958150145     537507077       -947304448
0x4006a6 <vulnerable+33>:       -88600  1166887167      -947304304      184
0x4006b6 <vulnerable+49>:       -24844288       -1924595713     -1991733179     137936838
0x4006c6 <vulnerable+65>:       12058688        -402653184      -450    1438894480
0x4006d6 <notcalled+1>: 1223002440      1209068675      1307592135      -1090502648
0x4006e6 <notcalled+17>:        4196458 184     -31660032       1170735103
0x4006f6 <notcalled+33>:        252     -1956254976     1665727557      1166756048
0x400706 <notcalled+49>:        -805222160      1006679567      -1959559584     1665727557
0x400716 <notcalled+65>:        1166756048      -805222160      1006679567      -1960870022
0x400726 <notcalled+81>:        1665727557      1166756048      -805222160      251704847
0x400736 <notcalled+97>:        -394018626      -389576447      -578    1166744299
0x400746 <notcalled+113>:       -798799620      -263877816      265290056       -1106313034
0x400756 <notcalled+129>:       -389576256      -606    33310083        1224492427
0x400766 <notcalled+145>:       -1958162333     21557317        11931600        -2022326140
0x400776 <notcalled+161>:       2751    -41883648       -913244161      254699203
0x400786:       33823   251658240       1463877663      -1992206783
0x400796 <__libc_csu_init+6>:   1096106495      630017108       2098798 764233813
0x4007a6 <__libc_csu_init+22>:  2098798 -158774957      836077897       -450278181
0x4007b6 <__libc_csu_init+38>:  149717832       66961736        -193048 -310032129
0x4007c6 <__libc_csu_init+54>:  521084532       132     -1991507968     -158774038
0x4007d6 <__libc_csu_init+70>:  1107265860      1222382847      1208075139      -361370823
```

The size of the buffer given is 100; which means ESP- IP Encapsulating Security Payload takes up a size of 100. The next space is taken up by EBP- which is the base pointer, it points to the top of the stack and when a function is called it is pushed, and popped on return- which occupies a space of 8 bytes(since we used a 64-bit processor).
Therefore, it is needed that we print 104 "X's" and concatenate to the address of 'notcalled()' which we acquired previously. As the stack follows LIFO- Last In First Out principle- the address is written backwards as

\xd5\x06\x40\x00\x00\x00\x00\x00 [3]. This statement is coded in python and is piped to oflow.c to get the required output.



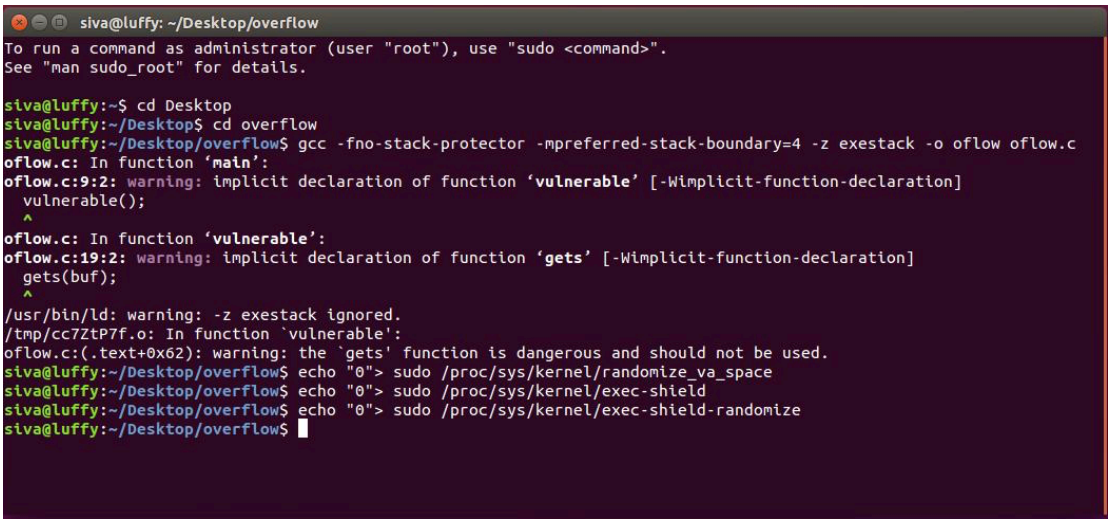
Counter Measure:

The usage of an unsafe function resulted in exploitation of the vulnerability. Using fgets() instead of gets() could be taken as a counter measure [3]. This can avoid buffer overflow attacks.

Task 4: Print current hostname to STDOUT

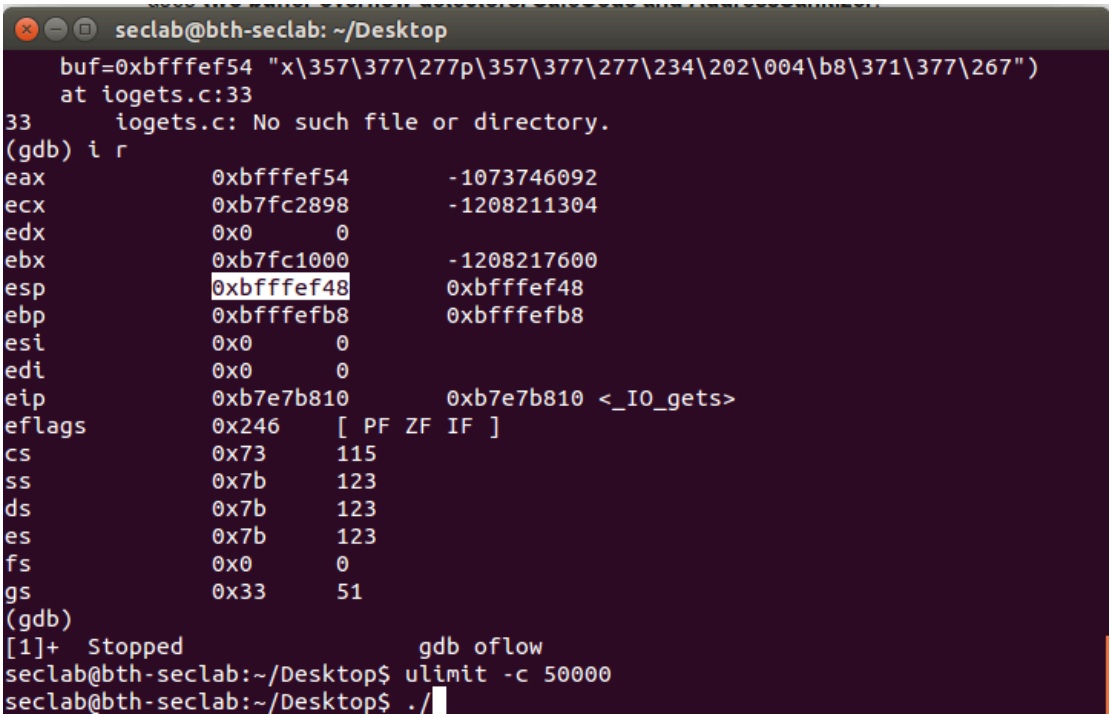
Vulnerability and Method:

This task is to print the hostname to STDOUT. Our first step is disable the stack protection in the system; which we do using the same commands as used for task 2.

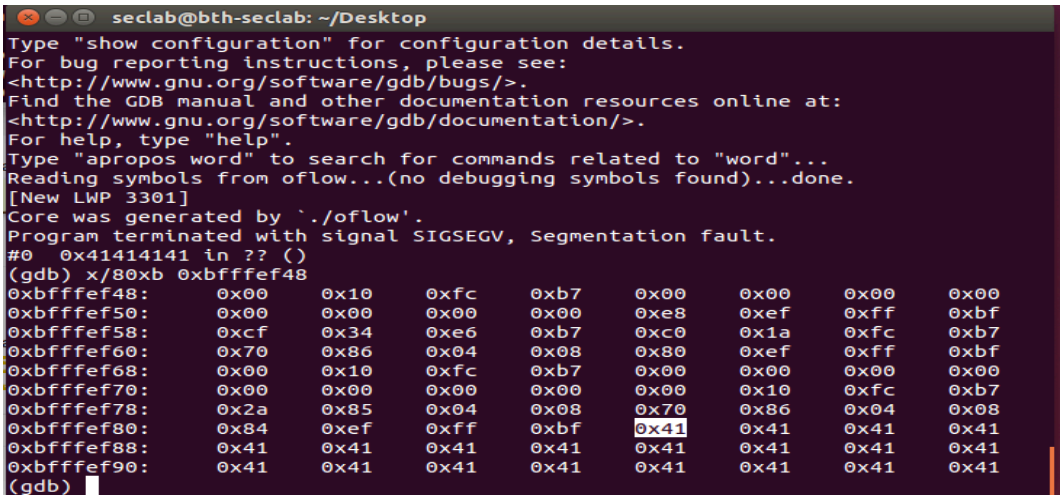


Next, we debug *oflow.c* using **gdb oflow**.
gdb is a GNU debugger, which helps in seeing what is going on inside another program while it executes- or what another program was doing at the moment it was crashed.
Execute list after the debugger gdb is opened, which will display the program code. The unsafe function gets() is displayed when **list 19** is executed. Next, we execute **run**.
This command indicates that **line 19** of the source code contains the breakpoint.

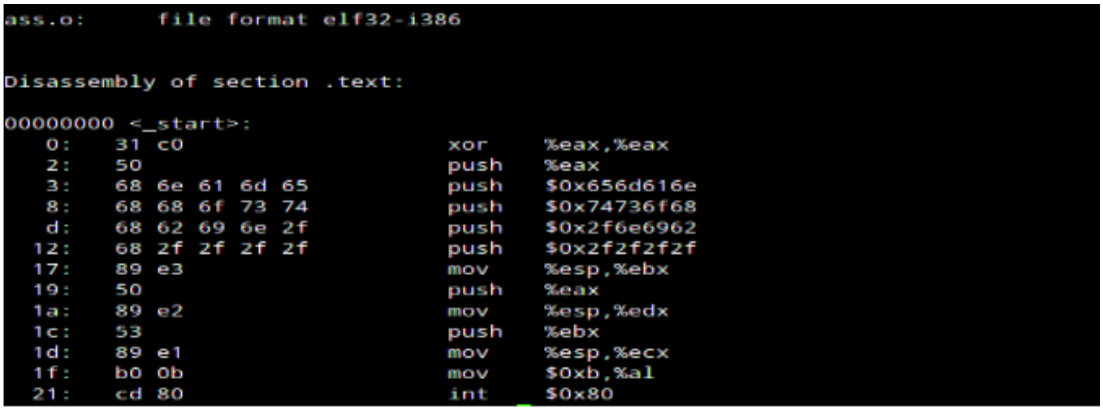
The command **i r** gives us information about registers which are currently available along with their address. We need **ESP register** whose address is given as **0xbfffef48.**



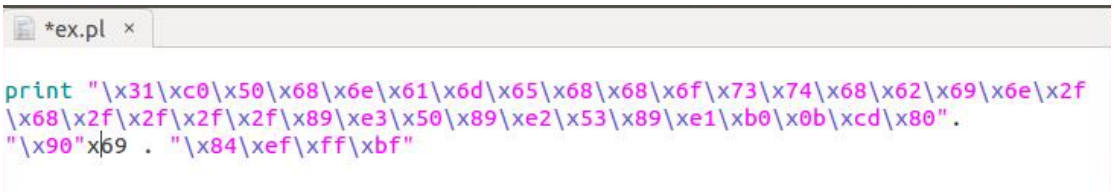
We exit the debugger using *Ctrl Z* as we have the information we need.
The core file is generated using the command **ulimit –c 5000.**
We now run oflow.c using a input string of
"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" .
This generates the core file, which will show us the return address to be overwritten. The command x/80xb 0xbfffef48 shows us the addresses and the values stored.
We search the core file for the hexadecimal value of X and it is noted as that address is the return address to be overwritten. The return address obtained is 0xbfffef84. On finding the starting address, we quit the debugger using Ctrl Z.



We need to generate an object file using **as -32 –o filename.o filename.s.** The object dump file is determined using **objdump –d filename.o.** This will give us the addresses of the hostname, stdbin and others for which an assembly code has to be written.



We create another file and write the code in perl language;



then pipe it to oflow.c using **perl ex.pl | ./oflow** .
Executing this command will give the hostname of the system, which is the required output.

Counter Measure:
Use fgets()-safe function; instead of gets() to avoid buffer overflow attacks.

Reflexion:

Our Lab Assignment 01- Source Code Analysis gave us a good overview about vulnerabilities. Our task 1 in the previous lab assignment was about heap buffer overflow attack, which made us get familiar with overflow attacks in general. We spent around 4 hours revising the concepts before starting this lab assignment. Our preparation for viva also came handy for this assignment.

Task 1: It was indeed a warm up task, as mentioned in the guidelines given to us. We spent 6 hours on this task. Initially, we divided the websites among ourselves as there are two of them and studied them individually. Later, we told the other group member about our grasp on the subject and then, we peer reviewed each other's work, just incase to avoid any mistakes. This process took 5 hours and it took an hour to document the task.

Task 2: We found this the most interesting task of all as we got to know the different flags, which could make and break an operating system's security mechanism. Having knowledge about the destruction buffer overflow can cause, we feel that we understood Ch-11 in Grey Hat Hacking in a better manner. We understood that as a computer user, it is good if we knowledgeable about the attacks that could be a harm in the future to us. We may be able to protect ourselves then. This task took around 6 hours to complete.

Task 3: The Appendix in the guidelines file was helpful apart from the textbook chapter. The syntax of the commands to execute notcalled() function were clearly given before hand and this made the task quite easy to complete. But there was some glitch in the output as the same command sometimes gave the correct output while sometimes; it didn't seem to work effectively. Nevertheless, we spent 7 hours on this task and gained knowledge about buffer overflow attacks making use of address of the function.

Task 4:
This task consumed around 16 hours to work. We had to reread the concepts in the textbook to understand the concepts properly. We were working on 64-bit computer for 3 hours in the beginning, but were unsuccessful in completing the task as we were unclear with the commands in gdb. We were not able to retrieve the address properly. Later, we worked for two hours in the lab and were successful in getting an output. The documentation took about two hours.

References

[1]. Common vulnerabilities and exposures list website (Date: November 24, 2014): http://cve.mitre.org/
[2]. Common weakness enumeration website (Date: November 24, 2014): http://cwe.mitre.org/
[3]. S. Harris, A. HARPER, C. Eagle, J. NESS, and M. LESTER, Gray Hat Hacking The Ethical Hackers Handbook. McGraw-Hill Publishing, 2011.
[4]. Aleph One, Smashing The Stack For Fun And Profit (Date: November 29, 2014): http://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf
[5]. The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2 http://www.intel.com/content/ www/us/en/processors/architectures-software-developer-manuals. html
[6]. gdb Quick Reference Card: http://users.ece.utexas.edu/~adnan/gdb-refcard. pdf
[7]. Debugging with GDB: http://sources.redhat.com/gdb/documentation/