WriteUps for Assignment 1

GDB challenges :

1.) So in this challenge a ELF executable file is provided . Now when we used gdb to get information about functions these all functions were there

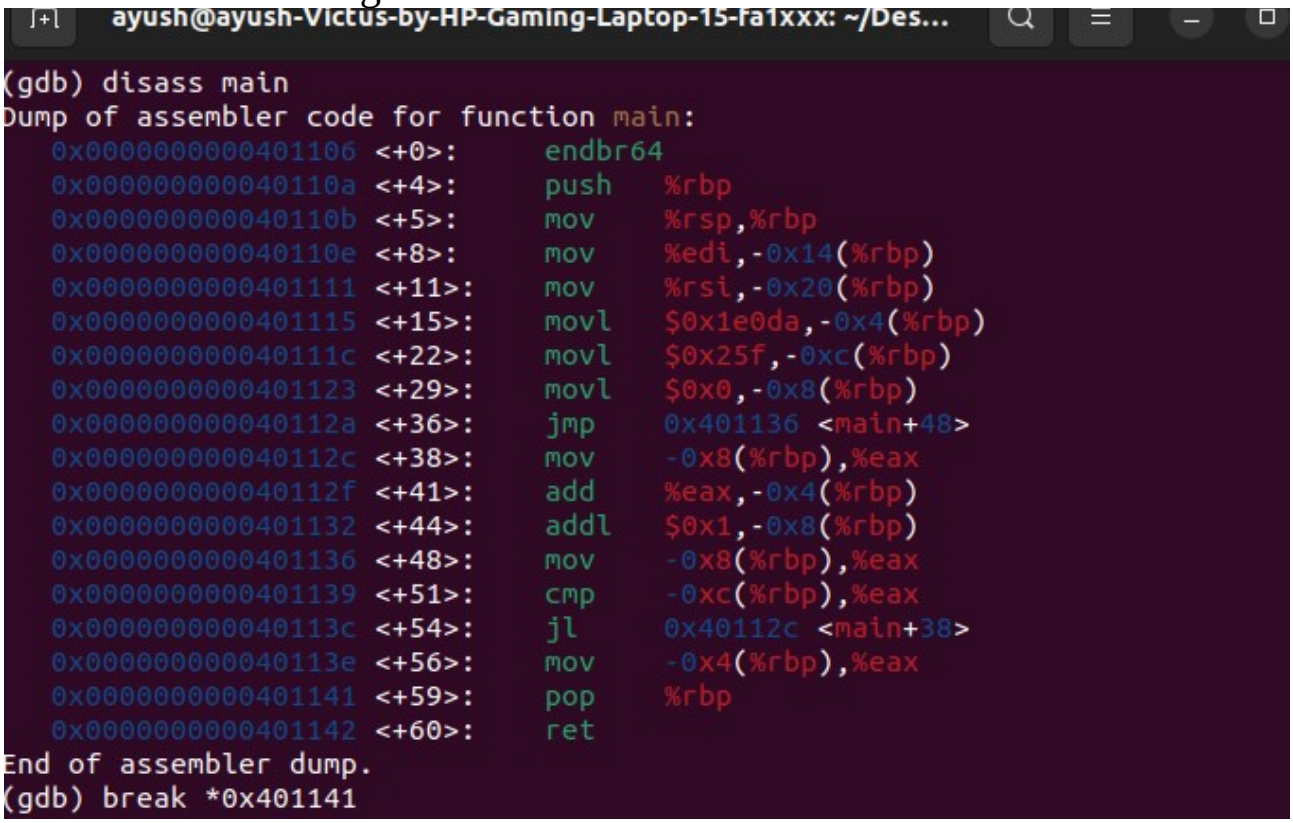
```
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from debugger0_a...
(No debugging symbols found in debugger0_a)
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x0000000000001000  _init
0x0000000000001030  __cxa_finalize@plt
0x0000000000001040  _start
0x0000000000001070  deregister_tm_clones
0x00000000000010a0  register_tm_clones
0x00000000000010e0  __do_global_dtors_aux
0x0000000000001120  frame_dummy
0x0000000000001129  main
0x0000000000001140  __libc_csu_init
0x00000000000011b0  __libc_csu_fini
0x00000000000011b8  _fini
(gdb) disass main
```

Now after disassembling the main function we get

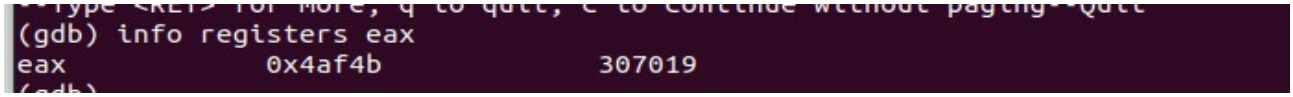
```
(gdb) disass main
Dump of assembler code for function main:
   0x0000000000001129 <+0>:     endbr64
   0x000000000000112d <+4>:     push   %rbp
   0x000000000000112e <+5>:     mov    %rsp,%rbp
   0x0000000000001131 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000000000001134 <+11>:    mov    %rsi,-0x10(%rbp)
   0x0000000000001138 <+15>:    mov    $0x86342,%eax
   0x000000000000113d <+20>:    pop    %rbp
   0x000000000000113e <+21>:    ret
End of assembler dump.
```

We can clearly see the the value 0x86342 is moved into eax register the value in decimal system is 549698 which is our flag.

2.) again in this challenge a ELF executable file is provided . Now when we used gdb to get information about functions and then we disassemble the main function to get



Now we make a breakpoint at the 56th line and then run the program when the execution stops at breakpoint we look for the value in eax register using info registers command



We can clearly see that our flag is 307019.

4.) again in this challenge a ELF executable file is provided . Now when we used gdb to get information about functions and then we disassemble the main function to get

```
(gdb) disass main
Dump of assembler code for function main:
   0x000000000040111c <+0>:     endbr64
   0x0000000000401120 <+4>:     push   %rbp
   0x0000000000401121 <+5>:     mov    %rsp,%rbp
   0x0000000000401124 <+8>:     sub    $0x20,%rsp
   0x0000000000401128 <+12>:    mov    %edi,-0x14(%rbp)
   0x000000000040112b <+15>:    mov    %rsi,-0x20(%rbp)
   0x000000000040112f <+19>:    movl   $0x28e,-0x4(%rbp)
   0x0000000000401136 <+26>:    movl   $0x0,-0x8(%rbp)
   0x000000000040113d <+33>:    mov    -0x4(%rbp),%eax
   0x0000000000401140 <+36>:    mov    %eax,%edi
   0x0000000000401142 <+38>:    call   0x401106 <func1>
   0x0000000000401147 <+43>:    mov    %eax,-0x8(%rbp)
   0x000000000040114a <+46>:    mov    -0x4(%rbp),%eax
   0x000000000040114d <+49>:    leave
   0x000000000040114e <+50>:    ret
End of assembler dump.
```

Now we can add two breakpoints one at 0x401142 and another one at 0x401147 to know the value stored in eax register before calling and after calling func1 respectively

```
(gdb) b *0x401142
Breakpoint 1 at 0x401142
(gdb) b *0x401147
Breakpoint 2 at 0x401147
(gdb) run
Starting program: /home/ayush/Desktop/picoctf/debugger0_d(1)
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0000000000401142 in main ()
(gdb) info registers eax
eax            0x28e               654
(gdb) continue
Continuing.

Breakpoint 2, 0x0000000000401147 in main ()
(gdb) info registers eax
eax            0x80c83e            8439870
(gdb) exit
A debugging session is active.
```

Now we can see the value in eax register before calling of functions (654) and after calling function (8439870) we can simply divide to get 12905 as our flag.

3.) again in this challenge a ELF executable file is provided . Now when we used gdb to get information about functions and then we disassemble the main function to get

```
0x00000000004011a8  _fini
(gdb) disass main
Dump of assembler code for function main:
   0x0000000000401106 <+0>:     endbr64
   0x000000000040110a <+4>:     push   %rbp
   0x000000000040110b <+5>:     mov    %rsp,%rbp
   0x000000000040110e <+8>:     mov    %edi,-0x14(%rbp)
   0x0000000000401111 <+11>:    mov    %rsi,-0x20(%rbp)
   0x0000000000401115 <+15>:    movl   $0x2262c96b,-0x4(%rbp)
   0x000000000040111c <+22>:    mov    -0x4(%rbp),%eax
   0x000000000040111f <+25>:    pop    %rbp
   0x0000000000401120 <+26>:    ret
End of assembler dump.
```

as we can see that the hexadecimal number is loaded in memory at 15th line so we set a breakpoint and then run the program

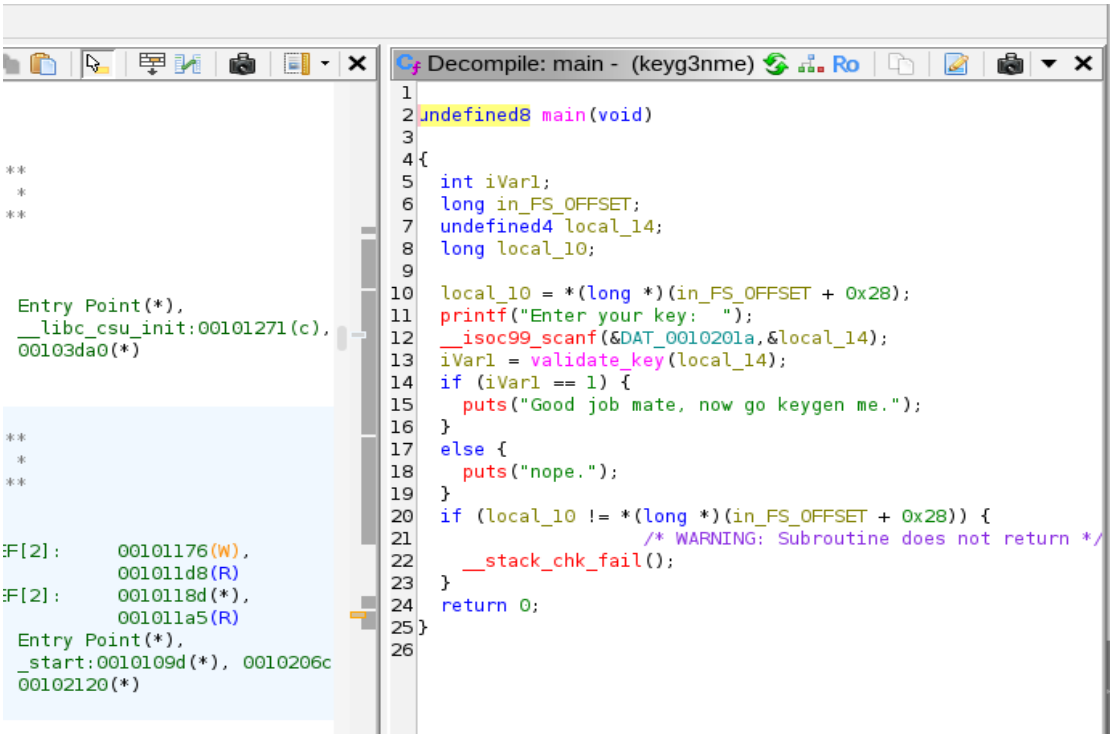
```
   0x0000000000401120 <+26>:    ret
End of assembler dump.
(gdb) b *0x40111c
Breakpoint 1 at 0x40111c
```

now we run the x/4xb command to get the required bytes

```
Breakpoint 1, 0x000000000040111c in main ()
(gdb) x/4xb $rbp-0x4
0x7fffffffdd0c: 0x6b    0xc9    0x62    0x22
(gdb)
```

Reversing challenges :

1.) After downloading and extracting the zip file we get a ELF executable named keyg3nme after opening the executable in ghidra we get our main function



Now we can see that user input is taken and is sent to a function validate_key , if the function returns 1 then we are done !

```
1
2 bool validate_key(int param_1)
3
4 {
5   return param_1 % 0x4c7 == 0;
6 }
7
```

This is the validate_key function we can see that this function just check input modulo 1223 so we can input any multiple of 1223 .

2.) After downloading and extracting the zip file we get another executable rev03 opening it with ghidra we get :

Decompile: main - (rev03)

```
ulong uVar2;
char local_98 [112];
long local_28;
int local_20;
int local_1c;

local_1c = 0;
puts("enter the magic string");
fgets(local_98,100,stdin);
sVar1 = strlen(local_98);
if (sVar1 < 0xc) {
  local_20 = 0;
  while( true ) {
    uVar2 = (ulong)local_20;
    sVar1 = strlen(local_98);
    if (sVar1 <= uVar2) break;
    local_1c = local_1c + local_98[local_20];
    local_20 = local_20 + 1;
  }
  if (local_1c == 1000) {
    local_28 = strcat_str();
    printf("flag is flag{");
    for (local_20 = 0; local_20 < 10; local_20 = local_20
      putchar((int)*(char *)(local_28 + local_20));
    }
    puts("}");
  }
  else {
    puts("wrong string\nNo flag for you.");
  }
}
else {
  puts("too long...sorry no flag for you!!!");
}
return 0;
}
```

So we can see that a string of maximum length 112 is taken as input and the length is stored in another variable called sVar1 , the length of string should be strictly smaller than 12 (0xc) now we can see that we are simply adding the ascii values of all charecters and it should sum out to 1000 , but the problem with this was that after pressing enter the system

also took a \n charecter its ascii value is 10 and hence the sum should be 990 we can use the string
“ccccccccc”

```
ayush@ayush-Victus-by-HP-Gaming-Laptop-15-fa1xxx:~/Desktop/picoctf$ ./rev03
enter the magic string
ccccccccc
flag is flag{!#&*/5<DMW}
ayush@ayush-Victus-by-HP-Gaming-Laptop-15-fa1xxx:~/Desktop/picoctf$
```

3.)