GNU Debugger and Password cracking

Adarsh Kumar Jain (2015UCP1547) B(5,6)

April 6, 2017

1 What is a debugger

A debugger is a program that runs other programs, allowing the user to exercise control over these programs, and to examine variables when problems arise.

GNU Debugger helps in getting information about the following:

- If a core dump happened, then what statement or expression did the program crash on?
- If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters?
- What are the values of program variables at a particular point during execution of the program?
- What is the result of a particular expression in a program?

2 Working of GDB

GDB allows us to run the program up to a certain point, then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

GDB uses a simple **command line interface**.

3 Debugging Symbol Table

Debugging Symbol Table maps instructions in the compiled binary program to their corresponding variable, function, or line in the source code. This mapping could be something like:

Program instruction item name, item type, original file, line number defined.

- A **symbol table** works for a particular version of the program if the program changes, a new table must be created.
- Debug builds are often larger and slower than retail (non-debug) builds; debug builds contain the symbol table and other ancillary information.
- If you wish to debug a binary program you did not compile yourself, you must get the symbol tables from the author.

4 Running GDB

To let **GDB** be able to read all that information line by line from the symbol table, it need to compile a source file **prog.c** with the **-g flag**:

gcc -g prog.c -o prog

5 Some Basic Commands for GDB

b main - Puts a breakpoint at the beginning of the program
b - Puts a breakpoint at the current line
b N - Puts a breakpoint at line N
b fn - Puts a breakpoint at the beginning of function "fn"
d N - Deletes breakpoint number N
info break - list breakpoints
r - Runs the program until a breakpoint or error
c - Continues running the program until the next breakpoint or error
f - Runs until the current function is finished
s - Runs the next line of the program
n - Like s, but it does not step into functions
p var - Prints the current value of the variable "var"
q - Quits gdb

6 Cracking a Password

Reuirements

1. **Disassembler**
   - Converts exe to assembly as best it can
   - Cannot always disassemble correctly
   - In general, it is not possible to assemble disassembly into working exe
2. **Debugger**
   - Must step through code to completely understand it
   - Labor intensive - lack of automated tools
3. **Hex Editor**
   - To patch (make changes to) exe file

7 Why Debugger Needed?

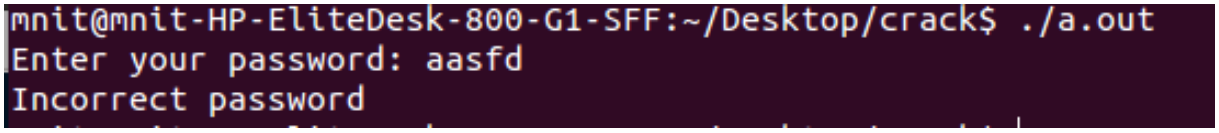
1. Disassembler gives static results:
   - ood overview of program logic.
   - But need to mentally execute program.
   - Difficult to jump to specific place in the code.
2. Debugger is dynamic:
   - Can set break points.
   - Can treat complex code as black box.
   - Not all code disassembles correctly.
3. Disassembler and debugger both required for any serious SRE task.

8 Example

This is the program which is taking a password as input and checks it by calling a function **check_password** which uses a function **strcmp** to compare the inputted password with the actual one. If the password matches, then **strcmp returns 0** and the function **returns 1** else function **returns 0**.

```c
#include<stdio.h>

void main()
{
    char s[50];

    printf("Enter your password: ");
    scanf("%s", s);

    int i = check_password(s);

    if(i==1)
        printf("Your password is correct\n");
    else
        printf("Incorrect password\n");
}

int check_password(char *s)
{
    if(strcmp(s,"asdfgh") == 0)
        return 1;
    else
        return 0;
}
```

When we run this program, it takes password as input. So we have to somehow bypass the password.



Bypassing the password:

1. To bypass the password, firstly we have to understand the program's flow and logic. For that we will create assembly for the given executable file using this command:

   objdump -D a.out

2. Now we have assembly for two functions which are in the source program **main** and **check_password**. We analyze these two functions assembly to understand the program for how it is checking password?

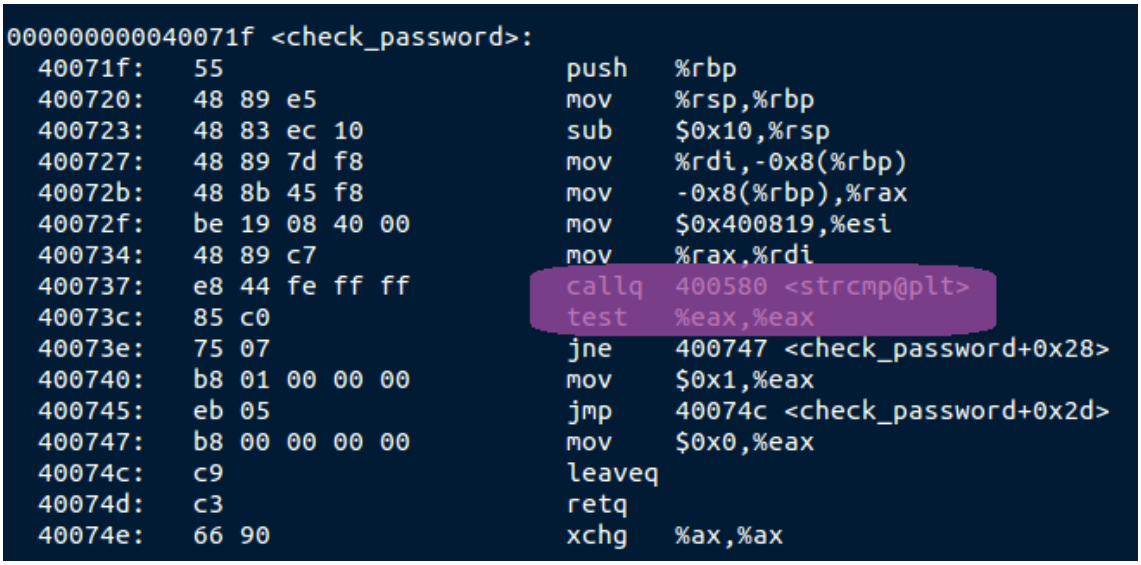
3. First we will analyze the **check_password** function.

   - In assembly of **check_password**, we can see at the address 400737, there is a call to strcmp function.

     400737: e8 44 fe ff ff callq 400580 strcmp@plt
     40073c: 85 c0 test %eax,%eax
     40073e: 75 07 jne 400747 ¡check_password+0x28

   - Then we have a test condition which is anding **eax** with **eax**. If these are not equal then it jumps to the address **400747** otherwise it continues the execution, move 1 into the eax to return this value and jumps to the address **40074c**

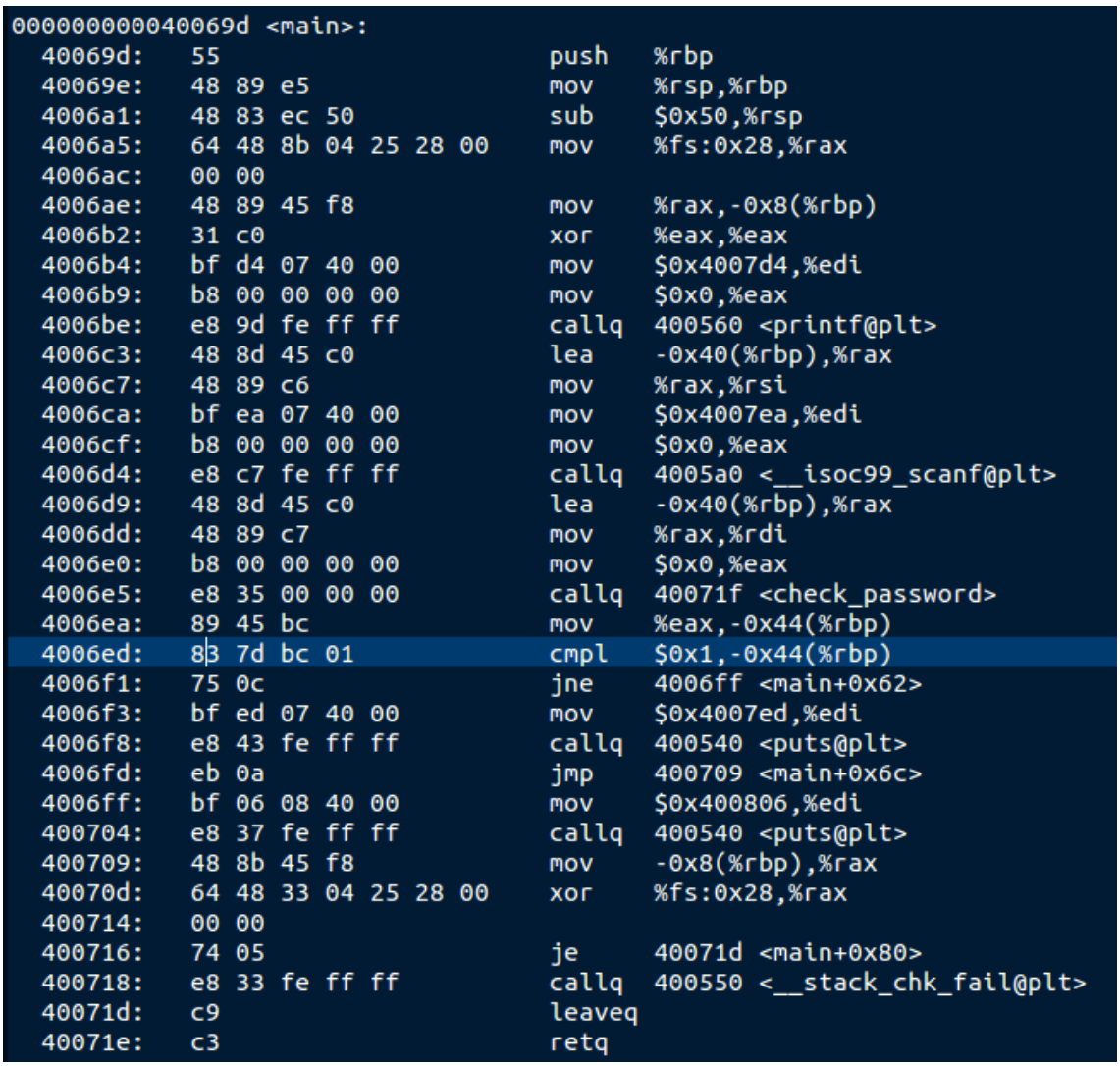
- From here, we may guess that it may be checking for password in **test** condition and if the password matches then it **puts 1** in the eax and returns, otherwise it **retruns 0**.



4. Now, we will analyze the assembly of main file.

   - In this file, at the address **4006ed**, we have a call to **check_password** function.
   - Then after that call it compares the eax value with 1. If it's not equal to 1 then it jumps to the location **4006ff** and prints a message, otherwise it continues the execution.
   - Now we knows that the returned value of function **check_password** is in eax and it compares that value with 1. As we have gussed that **check_password** function returns 1 if password successfully matches, so it might be possible that it is comparing the returned

values in main function.



```
000000000040069d <main>:
  40069d:   55                      push   %rbp
  40069e:   48 89 e5                mov    %rsp,%rbp
  4006a1:   48 83 ec 50             sub    $0x50,%rsp
  4006a5:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  4006ac:   00 00
  4006ae:   48 89 45 f8             mov    %rax,-0x8(%rbp)
  4006b2:   31 c0                   xor    %eax,%eax
  4006b4:   bf d4 07 40 00          mov    $0x4007d4,%edi
  4006b9:   b8 00 00 00 00          mov    $0x0,%eax
  4006be:   e8 9d fe ff ff          callq  400560 <printf@plt>
  4006c3:   48 8d 45 c0             lea    -0x40(%rbp),%rax
  4006c7:   48 89 c6                mov    %rax,%rsi
  4006ca:   bf ea 07 40 00          mov    $0x4007ea,%edi
  4006cf:   b8 00 00 00 00          mov    $0x0,%eax
  4006d4:   e8 c7 fe ff ff          callq  4005a0 <__isoc99_scanf@plt>
  4006d9:   48 8d 45 c0             lea    -0x40(%rbp),%rax
  4006dd:   48 89 c7                mov    %rax,%rdi
  4006e0:   b8 00 00 00 00          mov    $0x0,%eax
  4006e5:   e8 35 00 00 00          callq  40071f <check_password>
  4006ea:   89 45 bc                mov    %eax,-0x44(%rbp)
  4006ed:   83 7d bc 01             cmpl   $0x1,-0x44(%rbp)
  4006f1:   75 0c                   jne    4006ff <main+0x62>
  4006f3:   bf ed 07 40 00          mov    $0x4007ed,%edi
  4006f8:   e8 43 fe ff ff          callq  400540 <puts@plt>
  4006fd:   eb 0a                   jmp    400709 <main+0x6c>
  4006ff:   bf 06 08 40 00          mov    $0x400806,%edi
  400704:   e8 37 fe ff ff          callq  400540 <puts@plt>
  400709:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  40070d:   64 48 33 04 25 28 00    xor    %fs:0x28,%rax
  400714:   00 00
  400716:   74 05                   je     40071d <main+0x80>
  400718:   e8 33 fe ff ff          callq  400550 <__stack_chk_fail@plt>
  40071d:   c9                      leaveq
  40071e:   c3                      retq
```

5. When we run this command

   ndisasm -b 64 a.out

   on excutable file **a.out** it creates a assemlbly for the executable file **a.out** . In this file, we search for **test** instruction which is comparing eax with eax which we saw in the

check_password function.

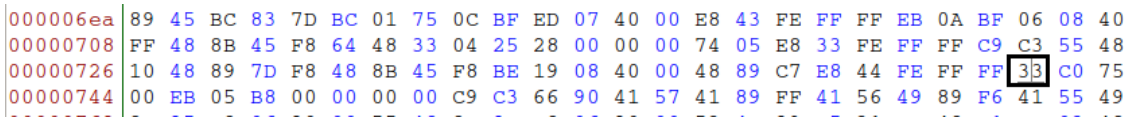
- It's on the address **0000073C**. After that, it jumps to the address **0x747** if result is not zero, that's mean **password is incorrect**.
- So somehow we have to make the **result always 0**, so that it continues the execution and returns 1.
- So we use **XOR operation** to ensure that result will always be zero and it always continues the execution and returns 1.
- That's how we **bypass the password**.

```
00000737  E844FEFFFF        call qword 0x580
0000073C  85C0              test eax,eax
0000073E  7507              jnz 0x747
00000740  B801000000        mov eax,0x1
00000745  EB05              jmp short 0x74c
00000747  B800000000        mov eax,0x0
0000074C  C9                leave
0000074D  C3                ret
```

6. We have a.out executable file opened in hex editor. In this editor, we can see that at the address **0x73C**, there is a hex-code **85** which is for **test** instruction in **check_password** function.

```
000006ea 89 45 BC 83 7D BC 01 75 0C BF ED 07 40 00 E8 43 FE FF FF EB 0A BF 06 08 40
00000708 FF 48 8B 45 F8 64 48 33 04 25 28 00 00 00 74 05 E8 33 FE FF FF C9 C3 55 48
00000726 10 48 89 7D F8 48 8B 45 F8 BE 19 08 40 00 48 89 C7 E8 44 FE FF FF 85 C0 75
00000744 00 EB 05 B8 00 00 00 00 C9 C3 66 90 41 57 41 89 FF 41 56 49 89 F6 41 55 49
```

7. So we changes that hex code **from 85 to 33** which is for **XOR** operation. Now we save this executable file with the name **patch**.



8. When we run this file, it **takes any password as input**. So we successfully bypass the password.