CSC 458 Reverse Engineer

In this lab we will focus on trying to reverse engineer a file. Explore the tools that are available to us.

First, download the file from the class web-page, the easiest way is to use the non-interactive network downloader, `wget`, that came with your linux installation:

```
wget http://www.cse.sdsmt.edu/ckarlsso/csc458/spring21/src/foobar
```

What is this?

The file has no extension this usually means it is an executable. We want to be sure, so the first thing we will do is to use the `file` command in an attempt to classify the file.

```
$ file foobar
foobar: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 2.6.24,
BuildID[sha1]=72922f8327afa289f1680c03499675863479489c, not stripped
```

Ok, so it is an ELF 64-bit LSB executable, that is at least a start. Next, as we downloaded the file our system might not recognize it as an executable, so we need to change the mode.

```
$ chmod +x foobar
```

Now we are able to run it:

```
$ ./foobar
Usage: <key>
$ ./foobar AAA-1234
Checking License: AAA-1234
WRONG!
```

Ok, we need more than this.

Bring out the guns

The first thing we are going to do is to explore the code using `gdb` it might not be installed in your version if not just go ahead and install it.

```
$ gdb ./foobar
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<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 ./foobar...
(No debugging symbols found in ./foobar)
(gdb)
```

Ok, we are now in the debugger (notice the new prompt), next step is trying to disassemble it. We don't know much, but we know that all programs have a main function, so we can start with that at least.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x00000000004005bd <+0>:     push   %rbp
   0x00000000004005be <+1>:     mov    %rsp,%rbp
   0x00000000004005c1 <+4>:     sub    $0x10,%rsp
   0x00000000004005c5 <+8>:     mov    %edi,-0x4(%rbp)
   0x00000000004005c8 <+11>:    mov    %rsi,-0x10(%rbp)
   0x00000000004005cc <+15>:    cmpl   $0x2,-0x4(%rbp)
   0x00000000004005d0 <+19>:    jne    0x400623 <main+102>
   0x00000000004005d2 <+21>:    mov    -0x10(%rbp),%rax
   0x00000000004005d6 <+25>:    add    $0x8,%rax
   0x00000000004005da <+29>:    mov    (%rax),%rax
   0x00000000004005dd <+32>:    mov    %rax,%rsi
   0x00000000004005e0 <+35>:    mov    $0x4006c4,%edi
   0x00000000004005e5 <+40>:    mov    $0x0,%eax
   0x00000000004005ea <+45>:    callq  0x400490 <printf@plt>
   0x00000000004005ef <+50>:    mov    -0x10(%rbp),%rax
   0x00000000004005f3 <+54>:    add    $0x8,%rax
   0x00000000004005f7 <+58>:    mov    (%rax),%rax
   0x00000000004005fa <+61>:    mov    $0x4006da,%esi
   0x00000000004005ff <+66>:    mov    %rax,%rdi
   0x0000000000400602 <+69>:    callq  0x4004b0 <strcmp@plt>
   0x0000000000400607 <+74>:    test   %eax,%eax
   0x0000000000400609 <+76>:    jne    0x400617 <main+90>
   0x000000000040060b <+78>:    mov    $0x4006ea,%edi
   0x0000000000400610 <+83>:    callq  0x400480 <puts@plt>
   0x0000000000400615 <+88>:    jmp    0x40062d <main+112>
   0x0000000000400617 <+90>:    mov    $0x4006fa,%edi
   0x000000000040061c <+95>:    callq  0x400480 <puts@plt>
   0x0000000000400621 <+100>:   jmp    0x40062d <main+112>
   0x0000000000400623 <+102>:   mov    $0x400701,%edi
   0x0000000000400628 <+107>:   callq  0x400480 <puts@plt>
   0x000000000040062d <+112>:   mov    $0x0,%eax
   0x0000000000400632 <+117>:   leaveq
   0x0000000000400633 <+118>:   retq
End of assembler dump.
(gdb)
```

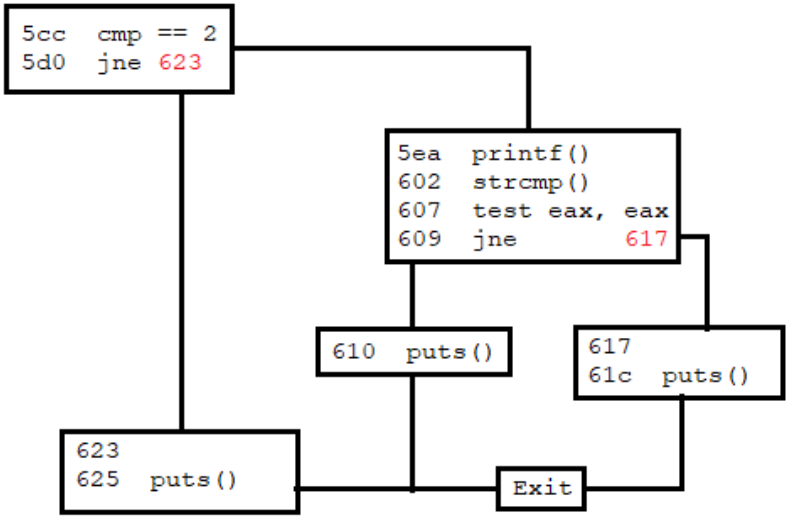
Shoot, silly AT&T syntax lets change the flavor to intel and try again.

```
(gdb) set disassembly-flavor intel
(gdb) disassemble main
Dump of assembler code for function main:
   0x00000000004005bd <+0>:     push   rbp
   0x00000000004005be <+1>:     mov    rbp,rsp
   0x00000000004005c1 <+4>:     sub    rsp,0x10
   0x00000000004005c5 <+8>:     mov    DWORD PTR [rbp-0x4],edi
   0x00000000004005c8 <+11>:    mov    QWORD PTR [rbp-0x10],rsi
   0x00000000004005cc <+15>:    cmp    DWORD PTR [rbp-0x4],0x2
   0x00000000004005d0 <+19>:    jne    0x400623 <main+102>
   0x00000000004005d2 <+21>:    mov    rax,QWORD PTR [rbp-0x10]
   0x00000000004005d6 <+25>:    add    rax,0x8
   0x00000000004005da <+29>:    mov    rax,QWORD PTR [rax]
   0x00000000004005dd <+32>:    mov    rsi,rax
   0x00000000004005e0 <+35>:    mov    edi,0x4006c4
   0x00000000004005e5 <+40>:    mov    eax,0x0
   0x00000000004005ea <+45>:    call   0x400490 <printf@plt>
   0x00000000004005ef <+50>:    mov    rax,QWORD PTR [rbp-0x10]
   0x00000000004005f3 <+54>:    add    rax,0x8
   0x00000000004005f7 <+58>:    mov    rax,QWORD PTR [rax]
   0x00000000004005fa <+61>:    mov    esi,0x4006da
   0x00000000004005ff <+66>:    mov    rdi,rax
   0x0000000000400602 <+69>:    call   0x4004b0 <strcmp@plt>
   0x0000000000400607 <+74>:    test   eax,eax
   0x0000000000400609 <+76>:    jne    0x400617 <main+90>
   0x000000000040060b <+78>:    mov    edi,0x4006ea
   0x0000000000400610 <+83>:    call   0x400480 <puts@plt>
   0x0000000000400615 <+88>:    jmp    0x40062d <main+112>
   0x0000000000400617 <+90>:    mov    edi,0x4006fa
   0x000000000040061c <+95>:    call   0x400480 <puts@plt>
   0x0000000000400621 <+100>:   jmp    0x40062d <main+112>
   0x0000000000400623 <+102>:   mov    edi,0x400701
   0x0000000000400628 <+107>:   call   0x400480 <puts@plt>
   0x000000000040062d <+112>:   mov    eax,0x0
   0x0000000000400632 <+117>:   leave
   0x0000000000400633 <+118>:   ret
End of assembler dump.
(gdb)
```

Ok, that is better, far easier to read. So what do we have? Well the instructions starts at `0x00000000004005bd` and ends at 0x`633`. We have two branches (`jne` at 5d0 and 609), we call external functions five times (`printf` at 5ea, `strcmp` at 602, then `puts` three times at 610, 61c and 628). If you are not sure about any of these commands use the man pages, but do it in the programmer's manual, by invoking `man 3 <cmnd>`. Ok, seems to me that `puts` is another way to write a string to `stdout`.

Next thing I usually do is to create a flow chart, so I can follow the structure.

```
0x00000000004005bd <+0>:     push   rbp
0x00000000004005be <+1>:     mov    rbp,rsp
0x00000000004005c1 <+4>:     sub    rsp,0x10
0x00000000004005c5 <+8>:     mov    DWORD PTR [rbp-0x4],edi
0x00000000004005c8 <+11>:    mov    QWORD PTR [rbp-0x10],rsi
0x00000000004005cc <+15>:    cmp    DWORD PTR [rbp-0x4],0x2
0x00000000004005d0 <+19>:    jne    0x400623 <main+102>
0x00000000004005d2 <+21>:    mov    rax,QWORD PTR [rbp-0x10]
0x00000000004005d6 <+25>:    add    rax,0x8
0x00000000004005da <+29>:    mov    rax,QWORD PTR [rax]
0x00000000004005dd <+32>:    mov    rsi,rax
0x00000000004005e0 <+35>:    mov    edi,0x4006c4
0x00000000004005e5 <+40>:    mov    eax,0x0
0x00000000004005ea <+45>:    call   0x400490 <printf@plt>
0x00000000004005ef <+50>:    mov    rax,QWORD PTR [rbp-0x10]
0x00000000004005f3 <+54>:    add    rax,0x8
0x00000000004005f7 <+58>:    mov    rax,QWORD PTR [rax]
0x00000000004005fa <+61>:    mov    esi,0x4006da
0x00000000004005ff <+66>:    mov    rdi,rax
0x0000000000400602 <+69>:    call   0x4004b0 <strcmp@plt>
0x0000000000400607 <+74>:    test   eax,eax
0x0000000000400609 <+76>:    jne    0x400617 <main+90>
0x000000000040060b <+78>:    mov    edi,0x4006ea
0x0000000000400610 <+83>:    call   0x400480 <puts@plt>
0x0000000000400615 <+88>:    jmp    0x40062d <main+112>
0x0000000000400617 <+90>:    mov    edi,0x4006fa
0x000000000040061c <+95>:    call   0x400480 <puts@plt>
0x0000000000400621 <+100>:   jmp    0x40062d <main+112>
0x0000000000400623 <+102>:   mov    edi,0x400701
0x0000000000400628 <+107>:   call   0x400480 <puts@plt>
0x000000000040062d <+112>:   mov    eax,0x0
0x0000000000400632 <+117>:   leave
0x0000000000400633 <+118>:   ret
```



Time to analyze

Let us insert a break point at the entry to the `main` (`break *main`), it makes it easier for us to step through the code. And after that we step into the execution (`run`), and the check the registers (`info registers`).

```
(gdb) break *main
Breakpoint 1 at 0x4005bd
(gdb) run
Starting program: /home/christer/src/Reverse/foobar

Breakpoint 1, 0x00000000004005bd in main ()
(gdb) info registers
rax            0x4005bd            4195773
rbx            0x400640            4195904
rcx            0x400640            4195904
rdx            0x7fffffffe008      140737488347144
rsi            0x7fffffffdff8      140737488347128
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf08      0x7fffffffdf08
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0xffffffffffffff8e  -114
r11            0x7ffff7de8fc0      140737351946176
r12            0x4004d0            4195536
r13            0x7fffffffdff0      140737488347120
r14            0x0                 0
r15            0x0                 0
rip            0x4005bd            0x4005bd <main>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb)
```

We can see the content of the registers, our instruction pointer (`rip`) points to `0x4005bd`, which is the beginning of the code. The Parity Flag (`PF`), Zero Flag (`ZF`), and Interrupt Enabled Flag (`EF`) are set. Just like on the ARM, they are bits in the `eflags` register. That is about all that we need to know right now. Let us use `si` to step a single instruction, and then check the registers again.

```
(gdb) si
0x00000000004005be in main ()
(gdb) i r
.............
rip            0x4005be            0x4005be <main+1>0
.............
```

Notice, that you can abbreviate `info registers` to just `i r`. It is also good to know that you can use both tab-complete and scroll through your entered commands using the arrow keys inside the `gdb`. Ok, from here we don't want to step into any function calls, so we will use [`n`]`ext[i]` instead. We are interested in what is happening at `0x5d0`, so all we need to do is keep pressing return (that will repeat the last command, which in our case was `ni`) until that instruction is executed.

```
(gdb) ni
0x00000000004005c1 in main ()
(gdb)
0x00000000004005c5 in main ()
(gdb)
0x00000000004005c8 in main ()
(gdb)
0x00000000004005cc in main ()
(gdb)
0x00000000004005d0 in main ()
(gdb)
0x0000000000400623 in main ()
(gdb)
```

We jumped to `0x623` from `0x5d0`, and if we press return two more times we pass through `0x628` and the usage message is presented (so this is the content of the `puts()` function call). As we didn't provide any argument, we can guess what the compare at the beginning of the code is all about. It checks that `argc == 2`. Let us provide an argument then. We don't need to step out of the `gdb`, all we have to do is type `run AAA-1234`, and once again step through the code using `ni`.

```
(gdb) run AAA-1234
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/christer/src/Reverse/foobar AAA-1234

Breakpoint 1, 0x00000000004005bd in main ()
(gdb) ni
0x00000000004005be in main ()
(gdb)
0x00000000004005c1 in main ()
(gdb)
0x00000000004005c5 in main ()
(gdb)
0x00000000004005c8 in main ()
(gdb)
0x00000000004005cc in main ()
(gdb)
0x00000000004005d0 in main ()
(gdb)
0x00000000004005d2 in main ()
(gdb)
```

Yes, we of course want to start from the beginning. This time we passed the branch and 0x5d0, and we can keep going until the next branch (`0x609`).

```
0x00000000004005d0 in main ()
(gdb)
0x00000000004005d2 in main ()
(gdb)
0x00000000004005d6 in main ()
(gdb)
0x00000000004005da in main ()
(gdb)
0x00000000004005dd in main ()
(gdb)
0x00000000004005e0 in main ()
(gdb)
0x00000000004005e5 in main ()
(gdb)
0x00000000004005ea in main ()
(gdb)
Checking License: AAA-1234
0x00000000004005ef in main ()
(gdb)
0x00000000004005f3 in main ()
(gdb)
0x00000000004005f7 in main ()
(gdb)
0x00000000004005fa in main ()
(gdb)
0x00000000004005ff in main ()
(gdb)
0x0000000000400602 in main ()
(gdb)
0x0000000000400607 in main ()
(gdb)
0x0000000000400609 in main ()
(gdb)
0x0000000000400617 in main ()
(gdb)
```

So, we jumped from `0x609` to `0x617`, that is because the `strcmp` returned something else than 0 (look in the man page if you are unfamiliar with `strcmp`). So, let us put a break before this section. The easiest way is to disassemble main again, type `break *`, and then mark the address (just the `0x0000000000400607`) and then press the mouse wheel to have gdb fill-in the text for us.

```
(gdb) disassemble main
Dump of assembler code for function main:
............
   0x0000000000400602 <+69>:    call   0x4004b0 <strcmp@plt>
   0x0000000000400607 <+74>:    test   eax,eax
   0x0000000000400609 <+76>:    jne    0x400617 <main+90>
```

```
.............
=> 0x0000000000400617 <+90>:    mov    edi,0x4006fa
.............
End of assembler dump.
(gdb) break *0x0000000000400607
Breakpoint 2 at 0x400607
(gdb)
```

BTW, notice how there is a little arow, showing our current location of execution (you can check that with `i r`). Time to execute the code again. As we have a second break point, and we are no longer interested in what is happening between the two, we write `continue` after we stop at the first one.

```
(gdb) run AAA-1234
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/christer/src/Reverse/foobar AAA-1234

Breakpoint 1, 0x00000000004005bd in main ()
(gdb) continue
Continuing.
Checking License: AAA-1234

Breakpoint 2, 0x0000000000400607 in main ()
(gdb)
```

We are about to execute: `0x0000000000400607 <+74>:    test   eax,eax`, and our next line is `0x0000000000400609 <+76>:    jne    0x400617 <main+90>`. So we will jump on not equal. As we saw in the beginning, AAA-1234 was not the correct key, so we expect to jump to 0x617. Let us confirm this, by using `ni` and step pass the branch:

```
(gdb) ni
0x0000000000400609 in main ()
(gdb)
0x0000000000400617 in main ()
(gdb)
0x000000000040061c in main ()
(gdb)
WRONG!
```

Ok, we know that `jne` branches on not equal, we have two options. We can manipulate `eax` (the lower 32 bit of the `rax` register), or we could manipulate the correct status flag. Let is manipulate the status flag. To do that we are going to re-run the code, and after we stopped at the second break point, will we step through the test, and right before we execute line `0x609`, set the `ZF`. The first thing we need to figure out is which bit in `eflag` that ZF is. Some research tells us it is the 6 bit in `eflags`. We are now ready to do this again. The command `set $eflags |= (1 << 6)` will set the sixth bit of eflags.

```
(gdb) run AAA-1234
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/christer/src/Reverse/foobar AAA-1234

Breakpoint 1, 0x00000000004005bd in main ()
(gdb) continue
Continuing.
Checking License: AAA-1234

Breakpoint 2, 0x0000000000400607 in main ()
(gdb) ni
0x0000000000400609 in main ()
(gdb) set $eflags |= (1 << 6)
(gdb) i r
............
rip            0x400609            0x400609 <main+76>
eflags         0x2c2               [ ZF SF IF ]
............
(gdb)
```

If everything is correct, our next line of execution should be `0x000000000040060b <+78>:` `mov edi,0x4006ea`, and after that the `puts()` for a correct key.

```
(gdb) ni
0x000000000040060b in main ()
(gdb)
0x0000000000400610 in main ()
(gdb)
Access Granted!
0x0000000000400615 in main ()
(gdb)
0x000000000040062d in main ()
(gdb)
0x0000000000400632 in main ()
(gdb)
0x0000000000400633 in main ()
(gdb)
__libc_start_main (main=0x4005bd <main>, argc=2, argv=0x7fffffffdfe8,
init=<optimized out>, fini=<optimized out>,
    rtld_fini=<optimized out>, stack_end=0x7fffffffdfd8) at
../csu/libc-start.c:342
342     ../csu/libc-start.c: No such file or directory.
(gdb)
```

YES! We did it. Well, we figured out how to get access. The trouble is we still don't know anything about the key. Let us attack that problem in the next lab!

Questions to answer:

1. I have the entire semester asked you to read the man-page for `wget`, maybe I should ask you something about it then. Does `wget` have the ability to finish a partially downloaded file?

2. Explain the `test eax, eax` in line `0x607`. What is in `eax`, and why is it not using `cmp eax, 0` here?

3. BTW, how do I remove a break point?

Submit solution as a PDF to the dropbox, no later than Friday 4/16 before class!