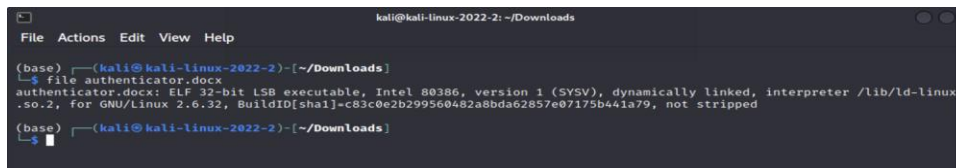


# A5 – Reverse Engineering

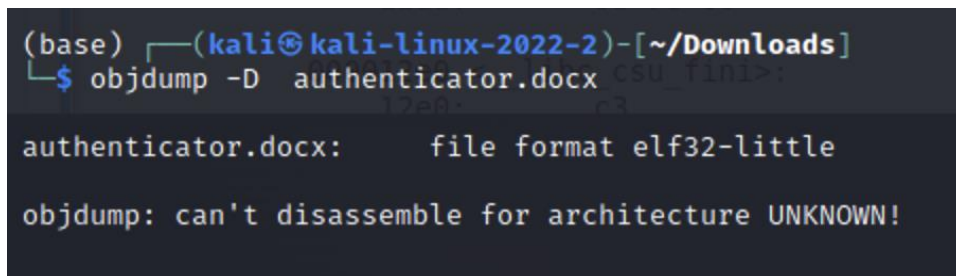
## i. Determine the type/format of the file

The first thing we can do is run the *file* command the results of which can be seen below:



```
kali@kali-linux-2022-2: ~/Downloads
File Actions Edit View Help
(base) └─(kali@kali-linux-2022-2)-[~/Downloads]
└─$ file authenticator.docx
authenticator.docx: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux
.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=c83c0e2b299560482a8bda62857e07175b441a79, not stripped
(base) └─(kali@kali-linux-2022-2)-[~/Downloads]
└─$
```

Because I am running an ARM based Kali Linux Machine (as well as my ARM based host environment) I ran into many problems related to the unknown state of the architecture that prevented the file from being disassembled and executed. This prevented me from finding the password until I was able to borrow someone else's computer in order to complete the assignment. Because of this major setback, I was unable to complete this assignment in full or on time.



```
(base) └─(kali@kali-linux-2022-2)-[~/Downloads]
└─$ objdump -D authenticator.docx
authenticator.docx:      file format elf32-little

objdump: can't disassemble for architecture UNKNOWN!
```

<https://unix.stackexchange.com/questions/119318/how-to-install-an-intel-binary-file-on-arm>

## ii. Reverse engineer the file to find the default passwords (backdoors) hidden

In order to find the default password in the file I used gdb to disassemble the binary and then set breakpoints and step through the assembly code lines while checking the values that are stored in each register. The following screenshots document this process:

The first breakpoint that I set was at the authenticate function using the gdb command **break authenticate**

The second breakpoint that I set was at the instruction line 0x080484d6 which calls the string compare function. This breakpoint can be seen below. When checking the values stored in some of the relevant registers, we can see that the password that was provided (1234) is stored in the eax register.

```

Breakpoint 12, 0x080484d6 in authenticate ()
(gdb) disass authenticate
Dump of assembler code for function authenticate:
0x080484ab <+0>: push %ebp
0x080484ac <+1>: mov %esp,%ebp
0x080484ae <+3>: sub $0x28,%esp
0x080484b1 <+6>: movl $0x0,-0xc(%ebp)
0x080484b8 <+13>: sub $0x8,%esp
0x080484bb <+16>: push 0x8(%ebp)
0x080484be <+19>: lea -0x20(%ebp),%eax
0x080484c1 <+22>: push %eax
0x080484c2 <+23>: call 0x8048360 <strcpy@plt>
0x080484c7 <+28>: add $0x10,%esp
0x080484ca <+31>: sub $0x8,%esp
0x080484cd <+34>: push $0x8048630
0x080484d2 <+39>: lea -0x20(%ebp),%eax
0x080484d5 <+42>: push %eax
=> 0x080484d6 <+43>: call 0x8048340 <strcmp@plt>
0x080484db <+48>: add $0x10,%esp
0x080484de <+51>: test %eax,%eax
0x080484e0 <+53>: je 0x80484fa <authenticate+79>
0x080484e2 <+55>: sub $0x8,%esp
0x080484e5 <+58>: push $0x8048639
0x080484ea <+63>: lea -0x20(%ebp),%eax
0x080484ed <+66>: push %eax
0x080484ee <+67>: call 0x8048340 <strcmp@plt>
0x080484f3 <+72>: add $0x10,%esp
0x080484f6 <+75>: test %eax,%eax
0x080484f8 <+77>: jne 0x8048501 <authenticate+86>
0x080484fa <+79>: movl $0x1,-0xc(%ebp)
0x08048501 <+86>: mov -0xc(%ebp),%eax
0x08048504 <+89>: leave
0x08048505 <+90>: ret
End of assembler dump.
(gdb) x/s $eax
0xffffcf88: "1234"
(gdb) x/s $esp
0xffffcf70: "\210\317\377\377\206\004\b@\332\377", <incomplete sequence \367>
(gdb)

```

The third breakpoint that I set was at the instruction line 0x080484ee which is the second call to the string compare function. This breakpoint can be seen below. We can see that the value of eax is still 1234 and the value that is pushed in the register 0x8048639 is "0x0xmain" which looked like a password to me.

```

Dump of assembler code for function authenticate:
0x080484ab <+0>: push %ebp
0x080484ac <+1>: mov %esp,%ebp
0x080484ae <+3>: sub $0x28,%esp
0x080484b1 <+6>: movl $0x0,-0xc(%ebp)
0x080484b8 <+13>: sub $0x8,%esp
0x080484bb <+16>: push 0x8(%ebp)
0x080484be <+19>: lea -0x20(%ebp),%eax
0x080484c1 <+22>: push %eax
0x080484c2 <+23>: call 0x8048360 <strcpy@plt>
0x080484c7 <+28>: add $0x10,%esp
0x080484ca <+31>: sub $0x8,%esp
0x080484cd <+34>: push $0x8048630
0x080484d2 <+39>: lea -0x20(%ebp),%eax
0x080484d5 <+42>: push %eax
0x080484d6 <+43>: call 0x8048340 <strcmp@plt>
0x080484db <+48>: add $0x10,%esp
0x080484de <+51>: test %eax,%eax
0x080484e0 <+53>: je 0x80484fa <authenticate+79>
0x080484e2 <+55>: sub $0x8,%esp
0x080484e5 <+58>: push $0x8048639
0x080484ea <+63>: lea -0x20(%ebp),%eax
0x080484ed <+66>: push %eax
=> 0x080484ee <+67>: call 0x8048340 <strcmp@plt>
0x080484f3 <+72>: add $0x10,%esp
0x080484f6 <+75>: test %eax,%eax
0x080484f8 <+77>: jne 0x8048501 <authenticate+86>
0x080484fa <+79>: movl $0x1,-0xc(%ebp)
0x08048501 <+86>: mov -0xc(%ebp),%eax
0x08048504 <+89>: leave
0x08048505 <+90>: ret
End of assembler dump.
(gdb) x/s $eax
0xffffcf88: "1234"
(gdb) x/s $ebp
0xffffcfab: "\330\317\377\377\205\004\b\215\322\377\377\375\367\312\301\367\240\024\374", <incomplete sequence \367>
(gdb) x/s $esp
0xffffcf70: "\210\317\377\377\206\004\b@\332\377", <incomplete sequence \367>
(gdb) x/s 0x8048340
0x8048340 <strcmp@plt>: "\377%\370\230\004\bh"
(gdb) x/s 0x80484fa
0x80484fa <authenticate+79>: "\307E\364\001"
(gdb) x/s 0x8
0x8: <error: Cannot access memory at address 0x8>
(gdb) x/s 0x8048639
0x8048639: "0x0xmain"
(gdb)

```

Sure enough, when we execute the file with the password "0x0xmain" we gain access to the secret part of the program. This can be seen in the screenshot below.

```
File Actions Edit View Help
Welcome, you have access to top secret part of the program!

(kali@kali)-[~/Downloads]
$
```

**iii. Modify the binary to execute the `/bin/sh` shell program when the user successfully authenticates**

This is the part of the assignment that I was unable to complete. Below is a compilation of links to resources that I found and was using.

<https://stackoverflow.com/questions/67993603/how-does-a-linux-c-c-system-command-work>

<https://stackoverflow.com/questions/14827894/launch-shell-with-inline-assembly>

We get the byte sequence of 2f62696e2f2f7368, which in ASCII is equal to `/bin//sh`

<https://axcheron.github.io/linux-shellcode-101-from-hell-to-shell/>

<https://stackoverflow.com/questions/65766170/assembly-code-to-shell-code-section-data-and-section-text-in-which-order>

**iv. Briefly explain checksums and calculate md5 and sha1 of the original and modified binaries and put those in the report**

Checksums are generated from running a cryptographic hash function on a file for the purpose of detecting errors that may have happened during its transmission or storage. Checksums are frequently used to verify data integrity but are not relied on to verify the authenticity of the data.

Below is a screenshot of the md5 and sha1 checksums of the original binary:

```
(kali@kali)-[~/Downloads]
$ md5sum authenticator.docx
69b72191324e806a484e3a52664b8380 authenticator.docx

(kali@kali)-[~/Downloads]
$ shasum authenticator.docx
b19badcab0a7bf759de1a310cccc72598b6720b6 authenticator.docx

(kali@kali)-[~/Downloads]
$
```

BELOW IS WORK:

Below is a screenshot of the assembly code for the authenticate function:

```

File Actions Edit View Help
8048497: 74 f2 je 804848b <frame_dummy+0xb>
8048499: 55 push %ebp
804849a: 89 e5 mov %esp,%ebp
804849c: 83 ec 14 sub $0x14,%esp
804849f: 50 push %eax
80484a0: ff d2 call *%edx
80484a2: 83 c4 10 add $0x10,%esp
80484a5: c9 leave
80484a6: e9 75 ff ff jmp 8048420 <register_tm_clones>

080484ab <authenticate>:
80484ab: 55 push %ebp
80484ac: 89 e5 mov %esp,%ebp
80484ae: 83 ec 28 sub $0x28,%esp
80484b1: c7 45 f4 00 00 00 movl $0x0,-0xc(%ebp)
80484b8: 83 ec 08 sub $0x8,%esp
80484bb: ff 75 08 push 0x8(%ebp)
80484be: 8d 45 e0 lea -0x20(%ebp),%eax
80484c1: 50 push %eax
80484c2: e8 99 fe ff ff call 8048360 <strcpy@plt>
80484c7: 83 c4 10 add $0x10,%esp
80484ca: 83 ec 08 sub $0x8,%esp
80484cd: 68 30 86 04 08 push $0x8048630
80484d2: 8d 45 e0 lea -0x20(%ebp),%eax
80484d5: 50 push %eax
80484d6: e8 65 fe ff ff call 8048340 <strcmp@plt>
80484db: 83 c4 10 add $0x10,%esp
80484de: 85 c0 test %eax,%eax
80484e0: 74 18 je 80484fa <authenticate+0x4f>
80484e2: 83 ec 08 sub $0x8,%esp
80484e5: 68 39 86 04 08 push $0x8048639
80484ea: 8d 45 e0 lea -0x20(%ebp),%eax
80484ed: 50 push %eax
80484ee: e8 4d fe ff ff call 8048340 <strcmp@plt>
80484f3: 83 c4 10 add $0x10,%esp
80484f6: 85 c0 test %eax,%eax
80484f8: 75 07 jne 8048501 <authenticate+0x56>
80484fa: c7 45 f4 01 00 00 movl $0x1,-0xc(%ebp)
8048501: 8b 45 f4 mov -0xc(%ebp),%eax
8048504: c9 leave
8048505: c3 ret

08048506 <main>:
8048506: 8d 4c 24 04 lea 0x4(%esp),%ecx
804850a: 83 e4 f0 and $0xffffffff,%esp
804850d: ff 71 fc push -0x4(%ecx)
8048510: 55 push %ebp
8048511: 89 e5 mov %esp,%ebp
8048513: 53 push %ebx
8048514: 51 push %ecx

```

Below is a screenshot of the assembly code of the authenticate function from inside gdb:

```

File Actions Edit View Help
[Inferior 1 (process 15463) exited normally]
(gdb) x/NFU 0x8048340
No symbol table is loaded. Use the "file" command.
(gdb) disassemble authenticate
Undefined command: "disassemble". Try "help".
(gdb) disassemble authenticate
Dump of assembler code for function authenticate:
0x080484ab <+0>: push %ebp
0x080484ac <+1>: mov %esp,%ebp
0x080484ae <+3>: sub $0x28,%esp
0x080484b1 <+6>: movl $0x0,-0xc(%ebp)
0x080484b8 <+13>: sub $0x8,%esp
0x080484bb <+16>: push 0x8(%ebp)
0x080484be <+19>: lea -0x20(%ebp),%eax
0x080484c1 <+22>: push %eax
0x080484c2 <+23>: call 0x8048360 <strcpy@plt>
0x080484c7 <+28>: add $0x10,%esp
0x080484ca <+31>: sub $0x8,%esp
0x080484cd <+34>: push $0x8048630
0x080484d2 <+39>: lea -0x20(%ebp),%eax
0x080484d5 <+42>: push %eax
0x080484d6 <+43>: call 0x8048340 <strcmp@plt>
0x080484db <+48>: add $0x10,%esp
0x080484de <+51>: test %eax,%eax
0x080484e0 <+53>: je 0x80484fa <authenticate+79>
0x080484e2 <+55>: sub $0x8,%esp
0x080484e5 <+58>: push $0x8048639
0x080484ea <+63>: lea -0x20(%ebp),%eax
0x080484ed <+66>: push %eax
0x080484ee <+67>: call 0x8048340 <strcmp@plt>
0x080484f3 <+72>: add $0x10,%esp
0x080484f6 <+75>: test %eax,%eax
0x080484f8 <+77>: jne 0x8048501 <authenticate+86>
0x080484fa <+79>: movl $0x1,-0xc(%ebp)
0x08048501 <+86>: mov -0xc(%ebp),%eax
0x08048504 <+89>: leave
0x08048505 <+90>: ret
End of assembler dump.
(gdb) ss

```

Below is a screenshot of the assembly code for the c++ function that spawns a shell: (this is what I have to integrate into the binary file to spawn a shell after the user authenticates)

a.out: file format mach-o arm64

Stackoverflow

AboutProductsFor Teams

Search...

Log in

Disassembly of section `__TEXT,__text`:

0000000100003f5c <\_\_Z9giveShellv>:

1000003f5c: fd 7b bf a9 stp x29, x30, [sp, #-16]!

1000003f60: fd 03 00 91 mov x29, sp

1000003f64: 00 00 00 90 adrp x0, 0x100003000 <\_\_Z9giveShellv+0x8>

1000003f68: 00 c0 3e 91 add x0, x0, #4016

1000003f6c: 0e 00 00 94 bl 0x100003fa4 <\_system+0x100003fa4>

1000003f70: fd 7b c1 a8 ldp x29, x30, [sp], #16

1000003f74: c0 03 5f d6 ret

0000000100003f78 <\_main>:

1000003f78: ff 83 00 d1 sub sp, sp, #32

1000003f7c: fd 7b 01 a9 stp x29, x30, [sp, #16]

1000003f80: fd 43 00 91 add x29, sp, #16

1000003f84: 08 00 80 52 mov w8, #0

1000003f88: e8 0b 00 b9 str w8, [sp, #8]

1000003f8c: bf c3 1f b8 stur wzr, [x29, #-4]

1000003f90: f3 ff ff 97 bl 0x100003f5c <\_\_Z9giveShellv>

1000003f94: e0 0b 40 b9 ldr w0, [sp, #8]

1000003f98: fd 7b 41 a9 ldp x29, x30, [sp, #16]

1000003f9c: ff 83 00 91 add sp, sp, #32

1000003fa0: c0 03 5f d6 ret

Disassembly of section `__TEXT,__stubs`:

0000000100003fa4 <\_\_stubs>:

1000003fa4: 10 00 00 b0 adrp x16, 0x100004000 <\_\_stubs+0x4>

1000003fa8: 10 02 40 f9 ldr x16, [x16]

1000003fac: 00 02 1f d6 br x16

Ask the compiler

program yourself, you can ask your compiler to emit assembly source. For  
piers use the `-S` switch.

g the GNU assembler, compiling with `-g -Wa,-alh` will give intermixed  
assembly listing, and `-ah` adds "high-level source" listing):

For Visual Studio, use `/FAsc`.

Peek into a binary

have a compiled binary,

How do you get assembler output C/C++ source in GCC?

Using GCC to produce readable assembly?

How to remove "noise" from GCC assembly output?

c++ for loop temporary variable us

Will a good C++ compiler optimize reference away?

NASM is pure assembly, but MASM high level Assembly?

which is the order that a compiler compiles.

How can I prove or disprove the efficiency of compilation?

Cost of virtuality and inheritance