FP3405 – Workshop Task – Static and Dynamic Analysis

Here we look at some of the tools to analyse a program, both statically (looking at the file on disk) and dynamically (at run time, in memory).

Before this task, follow the previous Workshop Task – Program Compilation Step by Step.

Simple Static Analysis

General tool – hexdump

Hexdump is a tool that allows us to look at the hexadecimal values that make up any file.
Let's make a simple text file

```
$ echo "This is my text string"
```

…this just prints the string to the screen ('Standard Output').

```
$ echo "This is my text string" > myfile
```

…this now directs ('>') the Standard Out to a file called 'myfile'.
If they file didn't exist, it will create the file. If it already existed, it will overwrite it.

Now we have a file; myfile. We can print it to screen with the `cat` command as we know it is text, but we can also look in this file and look at each byte:

```
$ hexdump myfile
```

…this shows you the file in hexadecimal.
By default, it will lump together 16 bits (2 bytes); that's four hexadecimal digits lumped together, as each hexadecimal digits represent 4 bits, or half a byte.

We can use the '`-C`' flag/switch to print it per byte, and also print an ASCII equivalent on the right hand side. Try it:

```
$ hexdump -C myfile            # ← That's a capital 'C'
```

You can use hexdump on any file, whether it is a text file, a jpg, png, gif, on documents and PDFs, on executable binaries, and so on.

Object tools – objdump, readelf, and nm

When you compile a program (e.g. `gcc source.c -o program`), then the output is known as object code. The object code could be a piece of a program (e.g. `test00.o`) that is yet to be linked with other pieces, a library (code for sharing between programs), or an executable program.

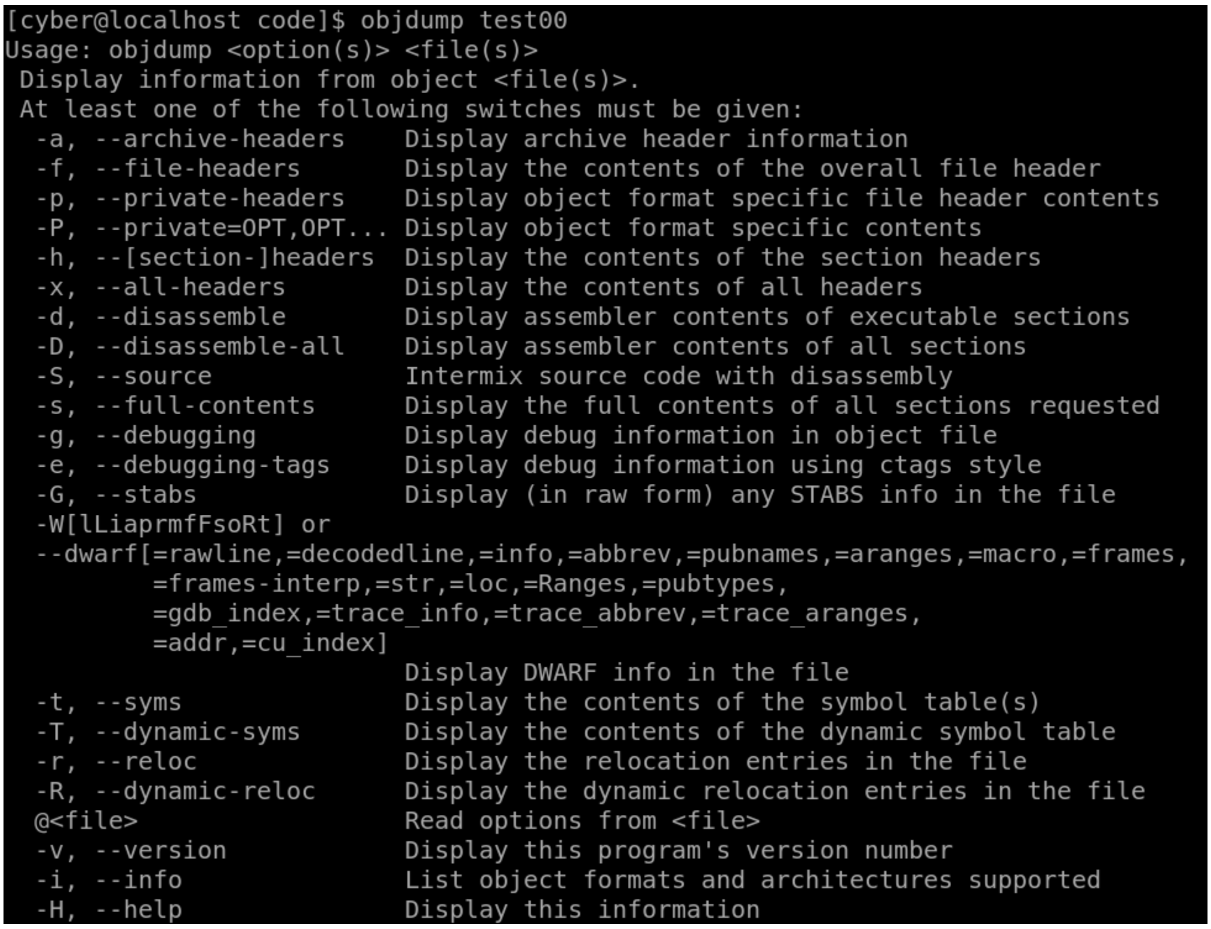
The tools, `objdump`, `readelf`, and `nm`, are for use on this 'object files'.

`objdump` – displays information from object files.

To run objdump, you need to pass it a flag/switch telling it what information you would like. E.g…

```
$ objdump program_name
```

…will not do anything yet, but it will give you a list of switches (options) to choose from:



```
[cyber@localhost code]$ objdump test00
Usage: objdump <option(s)> <file(s)>
 Display information from object <file(s)>.
 At least one of the following switches must be given:
  -a, --archive-headers    Display archive header information
  -f, --file-headers       Display the contents of the overall file header
  -p, --private-headers    Display object format specific file header contents
  -P, --private=OPT,OPT... Display object format specific contents
  -h, --[section-]headers  Display the contents of the section headers
  -x, --all-headers        Display the contents of all headers
  -d, --disassemble        Display assembler contents of executable sections
  -D, --disassemble-all    Display assembler contents of all sections
  -S, --source             Intermix source code with disassembly
  -s, --full-contents      Display the full contents of all sections requested
  -g, --debugging          Display debug information in object file
  -e, --debugging-tags     Display debug information using ctags style
  -G, --stabs              Display (in raw form) any STABS info in the file
  -W[lLiaprmfFsoRt] or
  --dwarf[=rawline,=decodedline,=info,=abbrev,=pubnames,=aranges,=macro,=frames,
          =frames-interp,=str,=loc,=Ranges,=pubtypes,
          =gdb_index,=trace_info,=trace_abbrev,=trace_aranges,
          =addr,=cu_index]
                           Display DWARF info in the file
  -t, --syms               Display the contents of the symbol table(s)
  -T, --dynamic-syms       Display the contents of the dynamic symbol table
  -r, --reloc              Display the relocation entries in the file
  -R, --dynamic-reloc      Display the dynamic relocation entries in the file
  @<file>                  Read options from <file>
  -v, --version            Display this program's version number
  -i, --info               List object formats and architectures supported
  -H, --help               Display this information
```

Let's choose one of these options. Let's 'disassemble' ('`-d`') the object file. Let's also do this on executable file you generated in the previous Workshop Task:

```
$ objdump -d test00
```

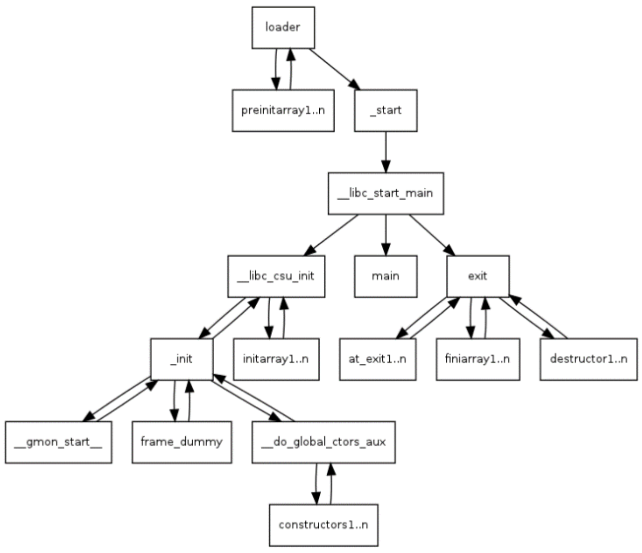
…this shows all the code, both the binary (in hexadec), and the dis-assembly code, that exists in the program.

The code is group together into functions. Can you find the main function? ('`<main>`')

Notice all the other functions which are linked into your program, other than main. (Remember the '**Program Start Up**' slide from previous slides?)

This diagram →

The linked-in extra functions get your program working when it is loaded into memory the first time.



Try the same '`-d`' flag on the object file:

```
$ objdump -d test00.o
```

Try some other flags from the plethora of options, on any object file. E.g:

```
$ objdump -t test00.o                  # ← The not yet linked code
$ objdump -t test00                    # ← The linked executable prog
```

…Try more options.

Note. On Linux and many Unix platforms, the object code etc is stored in a file called an **ELF** (Executable & Linkable Format) file, which is what we are dealing with here.
On other platforms there are different file types, e.g. **PE** (Portable Executable) on Windows, **COFF** (Common Object File Format) on Unix and embedded devices, **Mach-O** on Apple OS X.

Manual:

```
$ man objdump
```

`readelf` – displays information about ELF files.

```
$ readelf program_name
```

…will not do anything yet, but it will give you a list of switches (options) to choose from.

Let's choose one of these options. Let's print the header of the ('`-h`') the executable file:

```
$ readelf -h test00
```

Try the same '`-h`' flag on the object file. Try some other flags from the plethora of options, on any object file.

Manual:

```
$ man readelf
```

`nm` – lists symbols from object files.

```
$ nm test00.o
$ nm test00
```

Manual:

```
$ man nm
```

This concludes Simple Static Analysis. Static analysis is on the file or unchanging (non-running) thing; object code, executable, etc.

Simple Dynamic Analysis

We can run a program, but watch what it does when it runs. When we are watching and analysing what a program is doing when it is running (executing), this is called **Dynamic Analysis**.

`ltrace` and `strace` – a library and system call trace

We can watch what library calls a program make when it runs:

```
$ ltrace ./test00
```

…this time we are actually executing the program (`test00`) so we need to tell the computer where the program is, hence we need the '`./`' to indicate 'this directory'.

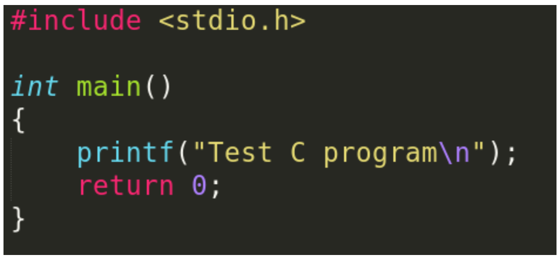
There are no libcalls (library calls) as the program simply returns:

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

Try a program with a printf() statement in it:

Call this test01.c

(`gcc test01.c -o test01`)



```
#include <stdio.h>

int main()
{
    printf("Test C program\n");
    return 0;
}
```

We can watch what library calls a program make when it runs:

```
$ ltrace ./test01
```

These are all the libcalls (library calls) the program did.
Notice we have a `puts()` libcall. This is what the printf() call that you do ultimately gets to call inside of the library.

We can watch what system calls a program makes when it runs:

```
$ strace ./test01
```

The program makes a lot of system calls.

Even if you have an empty program (`int main() { return 0}`), then you will still get a lot of system calls on Program Start Up. These system calls prepare the memory space / memory model of your process.

Taking a closer look at the output of the system calls made, we see:

```
==>   write(1, "Test C program\n", 15)
```

…this is the system call that is made by the library under the `puts()` library call, which is made by the `printf()` function call in your code.
The number '`1`' in the syscall means write to 'Standard Output', i.e. to the local screen.

gdb – The GNU Debugger

A debugger can run a program, halt/pause the program at a chosen location or under chosen conditions, and then present to the user an interface for looking inside of that program.

Basic operation is like this…

```
$ gdb ./program_name
```

…again, we are actually going to execute the program (`program_name`) so we need to tell the computer where the program is, hence we need the '`./`' to indicate 'this directory'.

Let's do it…

```
$ gdb ./test01
```

When we have done this, we are now inside the debugger (GDB). The program has been loaded into memory, ready to be run/executed, but is currently halted.

Before going any further, we can let the debugger we want the program to be halted when it reaches the `main()` function. This is called setting a breakpoint. To set the breakpoint, we give the command 'break' followed by the place we want the program to stop…

```
(gdb) break main
```

We can set more breakpoints if we so choose. We could give an addresses in memory instead of the name of the function (in this case main).

Now we can run the program. The program will run until it reaches the breakpoint, then it will stop and allow the user (you) to give further commands to analyse the program. Run…

```
(gdb) run
```

…the program has now been run, up to the breakpoint.

Let's analyse the program whilst it is halted/paused…

We can look at the values of all the registers inside the CPU for this program…

```
(gdb) info reg
```

We can use the '`x`' command to e**x**amine areas of memory etc. Let's examine 20 words of memory, from the top of an area of memory called the Stack. The top of the 'Call Stack', is pointed to by a register in the CPU called the SP (Stack Pointer). We can use this register to tell the e**x**amine command where to look…

```
(gdb) x /20 $sp
```

…this looks a bit messy. We can print the output in hexadecimal by using an extra 'x' after the number. Now it will print the output formatted as hexadecimal…

```
(gdb) x /20x $sp
```

Once we have finished looking in memory at this break point, we can allow the program to continue running by using the cont (continue) command…

```
(gdb) cont
```

Finally, you can quit GDB by pressing ctrl-D once, or by typing quit:

```
(gdb) quit
```

Feel free to run gdb again, set a breakpoint, and try different investigations of the memory or CPU registers etc.

This concludes Simple Dynamic Analysis. Dynamic analysis is on the running (dynamic) executable etc, whether running continually like with `ltrace` and `strace`, or temporarily halted with breakpoint such as with the `gdb` tool.

Self-assessment, and feedback to the lecturer

This was:	Easy | Fairly easy | So-so | Fairly tricky | Difficult
The material was:	Entirely new to me | some new / forgotten | Trivial

Common names for *nix non-alphanumeric characters:

( )	Parenthesis, parens
{ }	Braces
[ ]	Brackets, square brackets
-	Dash
/	Slash
\	Backslash, escape [sequence]
~	Tilde
#	Hash
$	String, dollar
|	Pipe
>	[Re-direct [standard] out]
<	[Re-direct [standard] in]
=	Assign, equals (non-mathematically)