Module 2.6 Compile code

A very useful data source is compiled files. This allows our pipeline to benefit from the sophisticated processing performed by each language's compiler. Thus, we don't need to reinvent the wheel by devising our own lexical analysis and parsing tools. Instead, we can obtain the processed source code data in the canonical form in which the compiler digests it. Many types of data files are amenable to this kind of processing. These include native code object files, virtual machine bytecode files, linked executable files, as well as statically and dynamically linked libraries. This approach works on many environments, for example for native code on Unix and Windows and in the Java ecosystem. We can see what's in native code object files with the nm command.

Here's a very simple C program called hello.c. It includes a header, it defines a global variable named global and a static variable named local, which is only visible within the file. There is also a local function named localfun, and a main function which is globally visible.

Main calls the fprintf function to print on the standard output: "hello, world: ". Next to "hello, world: " it prints the output of localfun. I compile the file hello.c with the cc command which stands for C Compiler. Next I run the nm command which stands for names or name list. I use it on the object file to see the symbols located in it. nm outputs a list of addresses, the type of each symbol, and the symbol name. Nm uses capital letters to denote the types of global elements. As we can see the first symbol of hello.o is a globally undefined symbol: fprintf. Then there is a global symbol named global, which is of type capital "C". "C" stands for common area; it is the way global variables are defined in C. The name is a remnant from old FORTRAN programs; a computer language developed in the 1950s and still in use for scientific computing. After global there is local, a symbol that corresponds to an uninitialized local function. Its type lowercase "b" stands for block starting with symbol. In this way local identifies a block that is initialized to plain zeroes. The following lowercase "t" symbol stands for text, which denotes code. It refers to the local function localfun. Furthermore, we have another capital "T" symbol for the global function main. The last symbol here is another undefined one: stdout.

We can also use the nm command on some executable files.

I compile again hello.c, but this time to an executable program instead of an object file. Running nm on the executable program hello we get many more symbols now. That's because the program I wrote has been compiled and linked together with the C library. For example, we see a capital "B" symbol, block starting symbol (bss) start, for the beginning address of all C uninitialized variables. Executable programs do not need their symbols to run. With the command called strip we can remove all symbols from an executable program. In this way I strip hello from its symbols. To verify that I run nm again on hello, and we get a warning that hello contains no symbols. Production systems are often shipped with their symbols stripped, in order to save disk space, or prevent reverse engineering. For instance, if I run nm on /bin/ls, we see that there are no symbols available.

As a complete example, let's now use the nm command to find which source code files contain the most global variables.

I build this command step by step on a directory containing all the compiled object files of the Apache web server. As a first step I run nm with the -o option, which means to precede each symbol with the filename where it occurs. I add as arguments to nm the output of the find command. find searches in the current directory hierarchy for files ending with the suffix ".o"; these are the object files. Then for each symbol found in the object files nm outputs the filename, the symbol type and its name. This was the first step in finding the file with the most global variables. As a second step I filter the output of nm to only list the uninitialized global data definitions. To do that I pipe the output of nm to grep capital "D", which represents the uninitialized global variables. Indeed we now see symbols such as dir_module, action_module and auth_module. As a third step I run again my pipeline but now I want to isolate the filenames. I use the cut command with the -d option to specify as the field separator the colon character. The filename is in the first field, so I also specify that with the -f option. As a result, we get a list of filenames: mod_dir, mod_actions and so on. The final step involves sorting the filenames by the number of their global variables. I run again the pipeline and pipe the output to sort, in order to group the filenames together. Then I run uniq -c to count the occurrences of each distinct filename. Finally, I sort the filenames in reverse numerical order. We see that http_main contains the most global variables: 37 (quite a small number). The following files contain either one or two variables each.

On Unix and on Windows most executable files are linked with dynamically linked libraries, also known as shared libraries. This allows the libraries to change independently of the programs that are linked to, and to save some disk space. Let's see how we can find information about which libraries are linked to which programs.

I run the ldd command on /bin/ls to list all dynamically linked libraries. As an output, for each library we get the corresponding system file it resolves to. To find the most used dynamic libraries, I first run ldd on all executable files in the bin directory. Then I pipe the output to sed to get rid of the left hand side libraries of the above list. I only want to keep the system files the libraries resolve to; these are the right-hand side parts of the list.

Similar to the previous example, I run sort to group the libraries together. Then I pipe the output to uniq -c to count their occurrences, and sort the libraries in reverse numerical order. With head we see the top ten used libraries. Three libraries share the first place: the Linux virtual dynamic shared object library (vDSO), the C standard library, and the dynamic linker library. These are used by all programs. The following libraries are used fewer times as we can see.

In the Java ecosystem we can use the javap command to look into Java class files.

Running javap on a jar file and specifying the path to a class we get the way this class is defined. The -c and -p arguments of javap are used to disassemble the file and list the class's private members. As we can see, the output of javap is much more detailed than the output of nm. That's because by default Java class files contain more information than compiled C programs.

On Windows we can use the dumpbin command in order to examine native code files.

As an example let's change to the Windows directory and use dumpbin to see what the Notepad program imports. I run dumpbin on the file notepad.exe specifying as an argument "/imports". On Windows arguments are specified with a slash rather than a minus. We see that Notepad imports various functions, such as RegistrySetValue and RegistryQueryValue. Furthermore, we can use the dumpbin command to see what a DLL exports. I change to the System32 directory where many DLLs are stored. There, I run dumpbin on wsock32.dll, the Windows Socket implementation. By specifying the "/exports" argument we see that it exports various functions, such as Accept and EnumProtocols.

This concludes our unit on fetching data from compiled code; stay with us!