Introduction to Reverse Engineering Software

Introduction to Reverse Engineering Software

Mike Perry

<mikepery@fscked.org>

Nasko Oskov

<nasko@netsekure.org>

Revision History	
Revision $Revision: 1.3 $	$Date: 2003/07/06 17:22:18 $

Abstract

This book is an attempt to provide an introduction to reverse engineering software under both Linux and Windows. Since reverse engineering is under legal fire, the authors figure the best response is to make the knowledge widespread. The idea is that since discussing specific reverse engineering feats is now illegal in many cases, we should then discuss general approaches, so that it is within every motivated user's ability to obtain information locked inside the black box. Furthermore, interoperability issues with closed-source proprietary systems are just plain annoying, and something needs to be done to educate more open source developers as to how to implement this functionality in their software.

[Note] **Note**

This book is actively being updated, and we are looking for a publisher. Please contact the authors if you are interested in helping to publish this book or know someone who would be.

[Note] **Note**

TO SLASHDOT READERS: Yes, this book is incomplete. Yes it has mistakes. Yes, we are working as hard as we can to fix them. Please email the authors directly rather than simply ranting/flaming on slashdot. We will take your comments into consideration, and will list you in the credits. We've already built up a large queue of fixes thanks to helpful emails.

Table of Contents

List of Figures

Next

Chapter 1. Introduction

Chapter 1. Introduction

Table of Contents

Prerequisites

This book is written at a level such that anyone who has taken an introductory computer science course (or has read the book Teach Yourself X in 21 days, where X is C or C++) should be able to understand all the material and work through all of the examples.

However, a data structures course (or a book that explains at least AVL trees, Hash Tables, Graphs, and priority queues), and a software engineering course (or even better, the book Design Patterns) would be very helpful not so much in understanding the following material, but more so in your ability to make the guesses and leaps needed to effectively reverse engineer software on your own.

What is reverse engineering?

Reverse engineering as this book will discuss it is simply the act of figuring out what software that you have no source code for does in a particular feature or function to the degree that you can either modify this code, or reproduce it in another independent work.

In the general sense, ground-up reverse engineering is very hard, and requires several engineers and a good deal of support software just to capture the all of the ideas in a system. However, we'll find that by using tools available to us, and keeping a good notebook of what's going on, we should be able to extract the information we need to do what matters: make modifications and hacks to get software that we do not have source code for to do things that it was not originally intended to do.

Why reverse engineer?

Answer: Because you can.

It comes down to an issue of power and control. Every computer enthusiast (and essentially any enthusiast in general) is a control-freak. We love the details. We love being able to figure things out. We love to be able to wrap our heads around a system and be able to predict its every move, and more, be able to direct its every move. And if you have source code to the software, this is all fine and good. But unfortunately, this is not always the case.

Furthermore, software that you do not have source code to is usually the most interesting kind of software. Sometimes you may be curious as to how a particular security feature works, or if the copy protection is really uncrackable, and sometimes you just want to know how a particular feature is implemented.

And we don't know about you, but to us, software that we don't have source code to just pisses us off. So we figure: screw it, lets do some damage. :)

It makes you a better programmer.

This book will teach you a large amount about how your computer works on a low level, and the better an understanding you have of that, the more efficient programs you can write in general.

To Learn Assembly Language.

If you don't know assembly language, at the end of this book you will literally know it inside-out. While most first courses and books on assembly language teach you how to use it as a programming language, you will get to see how to use C as an assembly language generation tool, and how to look at and think about assembly as a C program. This puts you at a tremendous advantage over your peers not only in terms of programming ability, but also in terms of your ability to figure out how the black box works. In short, learning this way will naturally make you a better reverse engineer. Plus, you will have the fine distinction of being able to answer the question "Who taught you assembly language?" with "Why, my C compiler, of course!"

Legal issues

FIXME: Pending... Research here and here (Also be aware of shrink-wrap licenses which forbid reverse engineering if you intend to publish results).

How to use this book

Learn the General Approach

This book is intended to give you an overview of Reverse Engineering under both UNIX and Windows. Most likely you will be initially interested in only one side or the other, but it is always a good idea to understand two different perspectives of the same idea. Even if you are not intending on ever using one of these two platforms now, the day will come when a particular program on one catches your eye, and you say to yourself, "Wouldn't it be neat if that ran on my OS? I wonder how I would go about doing that..." Knowing the general approach can allow you to rapidly adapt to new environments and paradigm shifts (ie you will be less thrown off when say, 64 bit architectures become prevalent, and less helpless when Palladium begins to see widespread usage).

The key insight is to think about how to use these tools and techniques to build as complete a map of your target application/feature as possible. Try not to focus on one tool or even one platform as the end-all-be-all of reverse engineering. Instead, try to focus on the process of information extraction, of fact gathering, and how each tool can give you a piece of the puzzle.

Read between the lines

This book is intentionally terse. We have a lot of material to cover, and the learning experience is intended to be hands-on rather than force-fed. We're not going to provide command summaries of every option of every tool. In fact, the most basic tools most likely will not even have output provided for them. The assumption is that the reader is either already familiar with these tools in the course of normal development/system usage, or is willing to play with the tools on their own.

On the contrary, we will not be skimping on the difficult material, such as learning assembly, or code modification techniques that are not as straightforward as simply running tools and looking at output. Hopefully you will still repeat or follow our example in your own projects.

Have a goal

None of the information in this book will be integrated into your thought process, or even retained, if you do not have some reason for reading it. Pick a program for which you want to figure out some small piece of it so that you can do something interesting. Maybe you want to replace a function call in an app to make it do something different, maybe you want to implement a particular feature of a program somewhere else, maybe you want to monitor all data before a program encrypts it and sends it across the network, or maybe you just want to cheat at your favorite multiplayer networked game.

Keep a notebook

Once you have this goal, define a map of your objectives. Get a multi-subject notebook, and divide it into sections. We suggest a Notes section, a Questions Section, an Active Hypotheses section, and an

Experiments section. Date all your entries, and save one section for a general diary, where you jot down a brief timeline of what you've done.

Every fact you pick up about your target application should make you feel a little triumphant. Write it down. Collect everything you can. These will come in handy, especially if the scope of your reversing effort is large.

Use the Scientific Method.

Remember 8th grade science class? Well guess what, it's relevant to reverse engineering. Essentially reverse engineering is a science in this sense (one could argue much more so than the rest of the slop-shod field of computer science itself). Consider every program you attack to be a system. You are performing educated guesses about that system, and then verifying these educated guesses with a look at the program behavior under a number of observational tools. To refresh your memory, the actual scientific method is an iteration over four steps:

1. Observe and describe a phenomenon or group of phenomena

   This is the first step. You notice something interesting in your application. An interesting behavior, a fluke, or just a sequence of events. Describe this well, trying to establish as many variables, unknowns, requisites and conditions as possible (using these terms in the general scientific sense, not the language syntactic sense - although we will see that these ideas are really parallel).
2. Formulate a hypothesis to explain these phenomena.

   Make an educated guess as to why this behavior occurred. Education is key. Hopefully you understand how software works at this point. And hopefully you have some data structures and pattern experience, or have a really good intuition for guessing how programs work. In any case, try to formulate a guess as to why these behavior are occurring. Some guidelines for this guess is that it should be comparable to the complexity of the feature. If it is something that can be implemented in one self-contained function, well then it should have a few variables that govern its behavior. Make predictions as to what will happen when these variables change.

   Sometimes, if you are looking at a large enough feature (or trying to determine a more complicated interaction), you need something more sophisticated than a simple function model. This still fits into this framework. If you have knowledge of finite state machines (which are basically just state transition diagrams) or push down automata (which are state transition diagrams with a stack, and are useful in language/grammar applications), you can go a long way to modeling more pieces of a system using the tools and techniques we introduce in this book. Just be sure to keep it in the back of your mind. If this paragraph scared you, don't worry. It is intended to give a name-drop overview of more formal methods you can use to model systems. The interested reader is encouraged to investigate these topics, but they won't come up in anything but large-scale reverse engineering efforts, usually involving protocols or parsing systems. (FIXME: we should consider devoting a chapter, appendix, or example to such a system)

You may also gain some information by taking a guess at the data structures used, or the design patterns employed. Also, this is usually only relevant to large scale reverse engineering efforts, but again, it fits into the framework and is worth mentioning.

3. Either try to use your hypothesis to predict new events, or attempt to find events that demonstrate your hypothesis is incorrect or incomplete.

   The latter is probably most useful, especially initially when trying to eliminate broad ranges of possibilities. (FIXME: Elaborate on this?)
4. Use your hypothesis to gain insight into the system, and perhaps even write some code.

   If you modify the environment of your program in certain ways, can you predict how this will affect it's behavior? Eventually the time will come to put your hypothesis to the ultimate test: If you code a component the way you think the original works, will your code do the original's job? If your goal is feature implementation details, it is probably a good idea to attempt to recode the feature and use a code modification technique to replace the original feature with yours. If your goal is modification, predict the action of the system under this modification, and verify it.

The most important thing to remember is that this is an iterative process. It converges on a solution through repetition of observation, guessing, testing, and predicting (coding). Initial loops through this process will start with major aspects of the system, and initial hypothesizing and testing should be done by actually using the application. You probably won't bring out the tools until the second or third iteration, and won't dive into the assembly until after that.

If you follow this procedure, you will narrow in on a solution relatively quickly. The most tempting thing is to skimp on the guess stage, and just test. This will get you limited results. You should try to structure your guesses and tests such that they eliminate large classes of possible operation first, and then zero in on the details. Note that nothing says these iterations have to be formal or written down. If your project is small, you can go through two or three iterations of the scientific method right in your head. But you still should be thinking about the system in this manner to be most effective.

If you notice that you have many different hypotheses about how the system works, build tests for them in order. If the feature you are after seems to depend on lots of variables, you should either narrow your focus, or try to develop a hierarchy or tree structure, with the variables that you suspect will effect the largest change at the top, and those that effect less change towards the leaves. Make predictions involving the largest variables first. If you find you have many different possible ways that your feature could work on different levels, again, organize a tree structure with the most likely way at the root, and then use a left branch to indicate that this hypthesis was incorrect, and a right branch to indicate that the general statement was correct. Typically, a correct hypothesis will lead to a whole new hypothesis tree, which you can either include or leave for another diagram, depending on the complexity.

Figure 1.1. Exploring a Hypothesis Space

Exploring a Hypothesis Space

Of course, you don't *have* to actually draw the tree, but it helps for more complicated scenarios, especially when you're dealing with many features at once. At the very least, this sort of organization should be going on in your head. Furthermore, you may find it useful to have more than two branches at certain points, but only if you can come up with a single test that somehow selects one outcome from several possible ones.

Most of the time for smaller efforts, you will probably only need one or two hypotheses that serve to simply point you in the right direction in the application, however, and you won't need to worry about doing anything complicated. Usually these will be something simple, like "This feature works with the help of such and such system library function(s)." Once you do a linker test to verify this and a trace to see where it calls this function, you're right where you need to be.

[Tip] **NOTE**

If you just haphazardly test without a battle plan, you will be in danger of performing unnecessary/irrelevant tests, or will waste your time looking at a lot of useless assembly code.

The Layout of the Book

The rest of the book is structured as a gradual decent from general to specific tools and techniques. We will first introduce tools that are used to gather information about the system/target as a whole. This will give us the information we need to form hypotheses about the next level of detail, namely, how our target is accomplishing various operations. We then can verify this using utilities that allow us a closer look at program behavior. From here, we then reapply the scientific method to hypothesize about the location and function of interesting segments of the program itself, based on which functions are being called from which regions of the program and in what manner. This should give us a hypothesis about the operation of our target in detail, which we then verify by looking at the assembly. (FIXME: Consider adding a "Form Your Hypothesis" section to each chapter).

From this point on, the game is all about how do we want to make use of this information. For this reason, various code modification and interception techniques are presented, including function insertion, RPC interception and buffer overflow techniques.

Prev Up Next

Chapter 2. The Compilation Process

Table of Contents

Intro

Compilation in general is split into roughly 5 stages: Preprocessing, Parsing, Translation, Assembling, and Linking.

Figure 2.1. The compilation Process

The compilation Process

All 5 stages are implemented by one program in UNIX, namely cc, or in our case, gcc (or g++). The general order of things goes gcc -> gcc -E -> gcc -S -> as -> ld.

Under Windows, however, the process is a bit more obfuscated, but once you delve under the MSVC++ front end, it is essentially the same. Also note that the GNU toolchain is available under Windows, through both the MinGW project as well as the Cygwin Project and behaves the same as under UNIX. Cygwin provides an entire POSIX compatibility layer and UNIX-like environment, where as MinGW just provides the GNU buildchain itself, and allows you to build native windows apps without having to ship an additional dll. Many other commercial compilers exist, but they are omitted for space.

The Compiler

Despite their seemingly disparate approaches to the development environment, both UNIX and Windows do share a common architectural back-end when it comes to compilers (and many many other things, as we will find out in the coming pages). Executable generation is essentially handled end-to-end on both systems by one program: the compiler. Both systems have a single front-end executable that acts as glue for essentially all 5 steps mentioned above.

The C Preprocessor

The preprocessor is what handles the logic behind all the # directives in C. It runs in a single pass, and essentially is just a substitution engine.

gcc -E

gcc -E runs only the preprocessor stage. This places all include files into your .c file, and also translates all macros into inline C code. You can add **-o file** to redirect to a file.

cl -E

Likewise, **cl -E** will also run only the preprocessor stage, printing out the results to standard out.

Parsing And Translation Stages

The parsing and translation stages are the most useful stages of the compiler. Later in this book, we will use this functionality to teach ourselves assembly, and to get a feel for the type of code generated by the compiler under certain circumstances. Unfortunately, the UNIX world and the Windows world diverge on their choice of syntax for assembly, as we shall see in a bit. It is our hope that exposure to both of these syntax methods will increase the flexibility of the reader when moving between the two environments. Note that most of the GNU tools do allow the flexibility to choose Intel syntax, should you wish to just pick one syntax and stick with it. We will cover both, however. (FIXME: Should we?)

gcc -S

gcc -S will take .c files as input and output .s assembly files in AT&T syntax. If you wish to have Intel syntax, add the option **-masm=intel**. To gain some association between variables and stack usage, use add **-fverbose-asm** to the flags.

gcc can be called with various optimization options that can do interesting things to the assembly code output. There are between 4 and 7 general optimization classes that can be specified with a -ON, where 0 <= N <= 6. 0 is no optimization (default), and 6 is usually maximum, although oftentimes no optimizations are done past 4, depending on architecture and gcc version.

There are also several fine-grained assembly options that are specified with the -f flag. The most interesting are -funroll-loops, -finline-functions, and -fomit-frame-pointer. Loop unrolling means to expand a loop out so that there are n copies of the code for n iterations of the loop (ie no jmp statements to the top of the loop). On modern processors, this optimization is negligible. Inlining functions means to effectively convert all functions in a file to macros, and place copies of their code directly in line in the calling function (like the C++ inline keyword). This only applies for functions called in the same C file as their definition. It is also a relatively small optimization. Omitting the frame pointer (aka the base pointer) frees up an extra register for use in your program. If you have more than 4 heavily used local variables, this may be rather large advantage, otherwise it is just a nuisance (and makes debugging much more difficult).

[Tip] **NOTE**

Since some of these get turned on by default in the higher optimization classes, it is useful to know that despite the fact that the manual page does not mention it explicitly, all of the -f options have -fno- equivalents. So -fno-inline-functions prevents function inlining, regardless of the -O option.

If you use -fverbose-asm, a non-inclusive list of compiler options is now printed at the top of the assembly output file. An annoying nuisance with gcc-3.x is that it enables many optimizations even at the -O0 level, making it difficult to generate hand-tuned asm from C. You can turn these off one by one using the above mentioned -fno- switch, however. Also one can write inline assembly to make sure that gcc will generate the code desired, but this should not be the preferred approach.

cl -S

Likewise, cl.exe has a -S option that will generate assembly, and also has several optimization options. Unfortunately, cl does not appear to allow optimizations to be controlled to as fine a level as gcc does. The main optimization options that cl offers are predefined ones for either speed or space. A couple of options that are similar to what gcc offers are:

-Ob<n> - inline functions (-finline-functions)
-Oy - enable frame pointer omission (-fomit-frame-pointer)

FIXME: Play with these.

Assembly Stage

The assembly stage is where assembly code is translated almost directly to machine instructions. Some minimal preprocessing, padding, and instruction reordering can occur, however. We won't concern ourselves with that too much, as it will become visible during disassembly, which is covered in the section Know Your Compiler

GNU as

as is the GNU assembler. It takes input as an AT&T or Intel syntax asm file and generates a .o object file.

MASM

MASM is the Microsoft assembler. FIXME: Where the hell is it?

Linking Stage

Both Windows and UNIX have similar linking procedures, although the support is slightly different. Both systems support 3 styles of linking, and both implement these in remarkably similar ways.

Static Linking

> Static linking means that for each function your program calls, the assembly to that function is actually included in the executable file. Function calls are performed by calling the address of this code directly, the same way that functions of your program are called.

Dynamic Linking

> Dynamic linking means that the library exists in only one location on the entire system, and the operating system's virtual memory system will map that single location into your program's address space when your program loads. The address at which this map occurs is not always guaranteed, although it will remain constant once the executable has been built. Functions calls are performed by making calls to a compile-time generated section of the executable, called the Procedure Linkage Table, PLT, or jump table, which is essentially a huge array of jump instructions to the proper addresses of the mapped memory. These structures will be discussed in Chapter 8, *Executable formats* and also in the Code Modification Chapter. (FIXME: Verify PLT on windows)

Runtime Linking

> Runtime linking is linking that happens when a program requests a function from a library it was not linked against at compile time. The library is mapped with dlopen() under UNIX, and LoadLibrary() under Windows, both of which return a handle that is then passed to symbol resolution functions (dlsym() and GetProcAddress()), which actually return a function pointer that may be called directly from the program as if it were any normal function. This approach is often

used by applications to load user-specified plugin libraries with well-defined initialization functions. Such initialization functions typically report further function addresses to the program that loaded them.

ld/collect2

ld is the GNU linker. It will generate a valid executable file. If you link against shared libraries, you will want to actually use what gcc calls, which is collect2. FIXME: Watch gcc -v for flags

link.exe

This is the MSVC++ linker. Normally, you will just pass it options indirectly via cl's -link option. However, you can use it directly to link object files and .dll files together into an executable. For some reason though, Windows requires that you have a .lib (or a .def) file in addition to your .dlls in order to link against them. The .lib file is only used in the interim stages, but the location to it must be specified on the -LIBPATH: option.

Prev Up Next
Chapter 1. Introduction Home Chapter 3. Gathering Info

Chapter 3. Gathering Info

Table of Contents

Now the fun begins. The first step to figuring out what is going on in our target program is to gather as much information as we can. Several tools allow us to do this on both platforms. Let's take a look at them.

System Wide Process Information

On Windows as on Linux, several applications will give you varying amounts of information about processes running. However, there is a one stop shop for information on both systems.

/proc

The Linux /proc filesystem contains all sorts of interesting information, from where libraries and other sections of the code are mapped, to which files and sockets are open where. The /proc filesystem contains a directory for each currently running process. So, if you started a process whose pid was 1337, you could enter the directory /proc/1337/ to find out almost anything about this currently running process. You can only view process information for processes which you own.

The files in this directory change with each UNIX OS. The interesting ones in Linux are: cmdline -- lists the command line parameters passed to the process cwd -- a link to the current working directory of the process environ -- a list of the environment variables for the process exe -- the link to the process executable fd -- a list of the file descriptors being used by the process maps -- VERY USEFUL. Lists the memory locations in use by this process. These can be viewed directly with gdb to find out various useful things.

Sysinternals Process Explorer

Sysinternals provides an all-around must-have set of utilities. In this case, Process Explorer is the functional equivalent of /proc. It can show you dll mapping information, right down to which functions are at which addresses, as well as process properties, which includes an environment tab, security id's, what files and objects are open, what the type of objects those handles are for, etc. It will also allow you to modify processes for which you have access to in ways that are not possible in /proc. You can close handles, change permissions, open debug windows, and change process priority.

Obtaining Linking information

The first step towards understanding how a program works is to analyze what libraries it is linked against. This can help us immediately make predictions as to the type of program we're dealing with and make some insights into its behavior.

ldd

ldd is a basic utility that shows us what libraries a program is linked against, or if its statically linked. It also gives us the addresses that these libraries are mapped into the program's execution space, which can be handy for following function calls in disassembled output (which we will get to shortly).

depends

depends is a utility that comes with the Microsoft SDK, as well as with MS Visual Studio. It will show you quite a bit about the linking information for a program. Not only will list dll's, but it will list which functions in those DLL's are being imported (used) by the current executable, and at what address they reside. This will come in very handy when we discuss code modification and interception.

Obtaining Function Information

The next step in reverse engineering is the ability to differentiate functional blocks in programs. Unfortunately, this can prove to be quite difficult if you aren't lucky enough to have debug information enabled. We'll discuss some of those techniques later.

nm

nm lists all of the local and library functions, global variables, and their addresses in the binary. However, it will not work on binaries that have been stripped with strip.

dumpbin.exe

Unfortunately, the closest thing Windows has to nm is dumpbin.exe, which isn't very great. The only thing it can do is essentially what depends already does: that is list functions used by this binary (dumpbin /imports), and list functions provided by this binary (dumpbin /exports). The only way a binary can export a function (and thus the only way the function is visible) is if that function has the __declspec(dllexport) tag next to it's prototype (FIXME: Verify).

Luckily, depends is so overkill, it often provides us with more than the information we need to get the job done.

Viewing Filesystem Activity

lsof

lsof is a program that lists all open files by the processes running on a system. An open file may be a regular file, a directory, a block special file, a character special file, an executing text reference, a library, a stream or a network file (Internet socket, NFS file or UNIX domain socket). It has plenty of options, but in its default mode it gives an extensive listing of the opened files. lsof does not come installed by default with most of the flavors of Linux/UNIX, so you may need to install it by yourself. On some distributions lsof installs in /usr/sbin which by default is not in your path and you will have to add it. An example output would be:

```
COMMAND     PID  USER    FD   TYPE     DEVICE     SIZE       NODE NAME
```

```
bash        101 nasko  cwd    DIR        3,2     4096    1172699 /home/nasko
bash        101 nasko  rtd    DIR        3,2     4096          2 /
bash        101 nasko  txt    REG        3,2   518140    1204132 /bin/bash
bash        101 nasko  mem    REG        3,2   432647     748736 /lib/ld-2.2.3.so
bash        101 nasko  mem    REG        3,2    14831    1399832
/lib/libtermcap.so.2.0.8
bash        101 nasko  mem    REG        3,2    72701     748743 /lib/libdl-2.2.3.so
bash        101 nasko  mem    REG        3,2  4783716     748741 /lib/libc-2.2.3.so
bash        101 nasko  mem    REG        3,2   249120     748742 /lib/libnss_compat-
2.2.3.so
bash        101 nasko  mem    REG        3,2   357644     748746 /lib/libnsl-2.2.3.so
bash        101 nasko    0u   CHR        4,5              260596 /dev/tty5
bash        101 nasko    1u   CHR        4,5              260596 /dev/tty5
bash        101 nasko    2u   CHR        4,5              260596 /dev/tty5
bash        101 nasko  255u   CHR        4,5              260596 /dev/tty5
screen      379 nasko  cwd    DIR        3,2     4096    1172699 /home/nasko
screen      379 nasko  rtd    DIR        3,2     4096          2 /
screen      379 nasko  txt    REG        3,2   250336     358394 /usr/bin/screen-
3.9.9
screen      379 nasko  mem    REG        3,2   432647     748736 /lib/ld-2.2.3.so
screen      379 nasko  mem    REG        3,2   357644     748746 /lib/libnsl-2.2.3.so
screen      379 nasko    0r   CHR        1,3              260468 /dev/null
screen      379 nasko    1w   CHR        1,3              260468 /dev/null
screen      379 nasko    2w   CHR        1,3              260468 /dev/null
screen      379 nasko    3r  FIFO        3,2             1334324
/home/nasko/.screen/379.pts-6.slack
startx      729 nasko  cwd    DIR        3,2     4096    1172699 /home/nasko
startx      729 nasko  rtd    DIR        3,2     4096          2 /
startx      729 nasko  txt    REG        3,2   518140    1204132 /bin/bash
ksmserver   794 nasko    3u  unix 0xc8d36580              346900 socket
ksmserver   794 nasko    4r  FIFO        0,6              346902 pipe
ksmserver   794 nasko    5w  FIFO        0,6              346902 pipe
ksmserver   794 nasko    6u  unix 0xd4c83200              346903 socket
ksmserver   794 nasko    7u  unix 0xd4c83540              346905 /tmp/.ICE-unix/794
mozilla-b  5594 nasko  144u  sock        0,0              639105 can't identify
protocol
mozilla-b  5594 nasko  146u  unix 0xd18ec3e0              639134 socket
mozilla-b  5594 nasko  147u  sock        0,0              639135 can't identify
protocol
mozilla-b  5594 nasko  150u  unix 0xd18ed420              639151 socket
```

Here is brief explanation of some of the abbreviations lsof uses in its output:

```
   cwd  current working directory
   mem  memory-mapped file
   pd   parent directory
   rtd  root directory
   txt  program text (code and data)
   CHR  for a character special file
   sock for a socket of unknown domain
   unix for a UNIX domain socket
   DIR  for a directory
   FIFO for a FIFO special file
```

It is pretty handy tool when it comes to investigating program behavior. lsof reveals plenty of information about what the process is doing under the surface.

> [Tip] **fuser**
>
> A command closely related to lsof is fuser. fuser accepts as a command-line parameter the name of a file or socket. It will return the pid of the process accessing that file or socket.

Sysinternals Filemon

The analog to lsof in the windows world is the Sysinternals Filemon utility. It can show not only open files, but reads, writes, and status requests as well. Furthermore, you can filter by specific process and operation type. A very useful tool.

Sysinternals Regmon

The registry in Windows is a key part of the system that contains lots of secrets. In order to try and understand how a program works, one definitely should know how the target interacts with the registry. Does it store configuration information, passwords, any useful information, and so on. Regmon from Sysinternals lets you monitor all or selected registry activity in real time. Definitely a must if you plan to work on any target on Windows.

Viewing Open Network Connections

So this is one of the cases where both Linux and Windows have the same exact name for a utility, and it performs the same exact duty. This utility is netstat.

netstat

netstat is handy little tool that is present on all modern operating systems. It is used to display network connections, routing tables, interface statistics, and more.

How can netstat be useful? Let's say we are trying to reverse engineer a program that uses some network communication. A quick look at what netstat displays can give us clues where the program connects and after some investigation maybe why it connects to this host. netstat does not only show TCP/IP connections, but also UNIX domain socket connections which are used in interprocess communication in lots of programs. Here is an example output of it:

Figure 3.1. Netstat output

```
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 slack.localnet:58705    egon:ssh                ESTABLISHED
tcp        0      0 slack.localnet:51766    gw.localnet:ssh         ESTABLISHED
tcp        0      0 slack.localnet:51765    gw.localnet:ssh         ESTABLISHED
tcp        0      0 slack.localnet:38980    clortho:ssh             ESTABLISHED
tcp        0      0 slack.localnet:58510    students:ssh            ESTABLISHED
Active UNIX domain sockets (w/o servers)
```

```
Proto RefCnt Flags       Type       State         I-Node Path
unix  5      [ ]         DGRAM                    68     /dev/log
unix  3      [ ]         STREAM     CONNECTED     572608 /tmp/.ICE-unix/794
unix  3      [ ]         STREAM     CONNECTED     572607
unix  3      [ ]         STREAM     CONNECTED     572604 /tmp/.X11-unix/X0
unix  3      [ ]         STREAM     CONNECTED     572603
unix  2      [ ]         STREAM                   572488
```

[Tip] **NOTE**

The output shown is from Linux system. The Windows output is almost identical.

As you can see there is great deal of info shown by netstat. But what is the meaning of it? The output is divided in two parts - Internet connections and UNIX domain sockets as mentioned above. Here is breifly what the Internet portion of netstat output means. The first column shows the protocol being used (tcp, udp, unix) in the particular connection. Receiving and sending queues for it are displayed in the next two columns, followed by the information identifying the connection - source host and port, destination host and port. The last column of the output shows the state of the connection. Since there are several stages in opening and closing TCP connections, this field was included to show if the connection is ESTABLISHED or in some of the other available states. SYN_SENT, TIME_WAIT, LISTEN are the most often seen ones. To see complete list of the available states look in the man page for netstat. FIXME: Describe these states.

Depending on the options being passed to netstat, it is possible to display more info. In particular interesting for us is the -p option (not available on all UNIX systems). This will show us the program that uses the connection shown, which may help us determine the behaviour of our target. Another use of this options is in tracking down spyware programs that may be installed on your system. Showing all the network connection and looking for unknown entries is invaluable tool in discovering programs that you are unaware of that send information to the network. This can be combined with the -a option to show all connections. By default listening sockets are not displayed in netstat. Using the -a we force all to be shown. -n shows numerical IP addesses instead of hostnames.

```
netstat -p as normal user
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 slack.localnet:58705    egon:ssh                ESTABLISHED -
tcp        0      0 slack.localnet:58766    winston:www             ESTABLISHED
5587/mozilla-bin
```

```
netstat -npa as root user
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 0.0.0.0:139             0.0.0.0:*               LISTEN
390/smbd
tcp        0      0 0.0.0.0:6000            0.0.0.0:*               LISTEN      737/X
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
78/sshd
tcp        0      0 10.0.0.3:58705          128.174.252.100:22      ESTABLISHED
```

```
13761/ssh
tcp        0      0 10.0.0.3:51766          10.0.0.1:22             ESTABLISHED
897/ssh
tcp        0      0 10.0.0.3:51765          10.0.0.1:22             ESTABLISHED
896/ssh
tcp        0      0 10.0.0.3:38980          128.174.252.105:22      ESTABLISHED
8272/ssh
tcp        0      0 10.0.0.3:58510          128.174.5.39:22         ESTABLISHED
13716/ssh
```

So this output shows that mozilla has established a connection with winston for HTTP traffic (since port is www(80)). In the second output we see that the SMB daemon, X server, and ssh daemon listen for incoming connections.

Gathering Network Data

Collecting network data is usually done with a program called sniffer. What the program does is to put your ethernet card into promiscuous mode and gather all the information that it sees. What is a promiscuous mode? Ethernet is a broadcast media. All computers broadcast their messages on the wire and anyone can see those messages. Each network interface card (NIC), as a hardcoded physical address called MAC (Media Access Control) address, which is used in the Ethernet protocol. When sending data over the wire, the OS specifies the destination of the data and only the NIC with the destination MAC address will actually process the data. All other NICs will disregard the data coming on the wire. When in promiscuous mode, the card picks up all the data that it sees and sends it to the OS. In this case you can see all the data that is flowing on your local network segment.

[Tip] **Disclaimer**

Switched networks eliminate the broadcast to all machines, but sniffing traffic is still possible using certain techniques like ARP poisoning. (FIXME: link with section on ARP poisoning if we have one.)

Several popular sniffing programs exist, which differ in user interface and capabilities, but any one of them will do the job. Here are some good tools that we use on a daily basis:

- ethereal - one of the best sniffers out there. It has a graphical interface built with the GTK library. It is not just a sniffer, but also a protocol analyzer. It breaks down the captured data into pieces, showing the meaning of each piece (for example TCP flags like SYN or ACK, or even kerberos or NTLM headers). Furthermore, it has excellent packet filtering mechanisms, and can save captures of network traffic that match a filter for later analysis. It is available for both Windows and Linux and requires (as almost any sniffer) the pcap library. Ethereal is available at www.ethereal.com and you will need libpcap for Linux or WinPcap for Windows.
- tcpdump - one of the first sniffing programs. It is a console application that prints info to the screen. The advantage is that it comes by default with most Linux distributions. Windows version is available as well, called WinDump.
- ettercap - also a console based sniffer. Uses the ncurses library to provide console GUI. It has built in ARP poisoning capability and supports plugins, which give you the power to modify data on the fly. This makes it very suitable for all kinds of Man-In-The-Middle attacks (MITM), which will we will describe in chapter (FIXME: link). Ettercap isn't that great a sniffer, but nothing prevents you from using its ARP poisoning and plugin features while also running a more powerful sniffer such as ethereal.

Now that you know what a sniffer is and hopefully learned how to use basic functionality of your favorite one, you are all set to gather network data. Let's say you want to know how does a mail client authenticate and fetch messages from the server. Since the protocol in use is POP3, we should instruct ethereal (our sniffer of choice) to capture traffic only destined to port 110 or originating from port 110. If you have a lot of machines checking mail at the same time on a network with a hub, you might want to restrict the matching only to your machine and the server you are connecting to. Here is an example of captured packet in ethereal: Ethereal breaks down the packet for us, showing what each part of the data means. For example, 1 shows us the Ethernet level information, such as source and destination MAC address. Also meaning of each bit in flag values are explained. Looking at the TCP header information, it says that the ... ... bits are set and the rest are not. Using packet captures, one can trace the flow of a protocol to better understand how an application works, or even try to reverse engineer the protocol itself if unknown.

Prev Up Next
Chapter 2. The Compilation Process Home Chapter 4. Determining Program Behavior

Chapter 4. Determining Program Behavior

Table of Contents

There are a couple of tools that allow us to look into program behavior at a more closer level. Lets look at some of these:

Tracing System Calls

This section is really only relevant for to our efforts under UNIX, as Windows system calls change regularly from version to version, and have unpredictable entry points.

strace/truss(Solaris)

These programs trace system calls a program makes as it makes them.

Useful options:

1. -f (follow fork)
2. -ffo filename (output trace to filename.pid for forking)
3. -i (Print instruction pointer for each system call)

Tracing Library Calls

Now we're starting to get to the more interesting stuff. Tracing library calls is a very powerful method of system analysis. It can give us a *lot* of information about our target.

ltrace

This utility is extremely useful. It traces ALL library calls made by a program.

Useful options:

1. -S (display syscalls too)
2. -f (follow fork)
3. -o filename (output trace to filename)
4. -C (demangle C++ function call names)
5. -n 2 (indent each nested call 2 spaces)
6. -i (prints instruction pointer of caller)
7. -p pid (attaches to specified pid)

API Monitor

API Monitor is incredible. It will let you watch .dll calls in real time, filter on type of dll call, view

Prev Up Next
Chapter 3. Gathering Info Home Chapter 5. Determining Interesting Functions

Chapter 5. Determining Interesting Functions

Table of Contents

Clearly without source code, we can't possibly hope to understand all of sections of an entire program. So we have to use various methods and guess work to narrow down our search to a couple of key functions.

Reconstructing function & control information

The problem is that first, we must determine what portions of the code are actually functions. This can be difficult without debugging symbols. Fortunately, there are a couple of utilities that make our lives easier.

objdump

Objdump's most useful purpose is to disassemble a program with the -d switch. Lacking symbols, this output is a bit more cryptic. The -j option is used to specify a segment to disassemble. Most likely we will want .text, which is where all the program code lies.

Note that the leftmost column of objdump contains a hex number. This is in fact the actual address in memory where that instruction is located. Its binary value is given in the next column, followed by its mnemonic.

objdump -T will give us a listing of all library functions this program calls.

disasm.pl

Steve Barker wrote a neat little perl script that makes objdump much more legible in the event that symbols are not included. The script has since been extended and improved by myself and Nasko Oskov.

It now makes 3 passes through the output. The first pass builds a symbol table of called and jumped-to locations. The second pass finds areas between two rets, and inserts them into the symbol table as "unused" functions. The third pass prints out the nicely labeled output, and prints out a function call tree. Usage:

```
./disasm /path/to/binary > binary.asminfo
```

There are/will be few command line options to the utility. Now --graph is supported. It will generate a file called call_graph that contains definition that can be used with a program called dot to generate visual representation of the call graph.

Note: Unused functions just mean that that function wasn't called DIRECTLY. It is still possible that a function was called through a function pointer (ie, main is called this way)

Consider the objective

Ok, so now we're getting ready to get really down and dirty. The first step to finding what you are looking for is to know what you are looking for. Which functions are 'interesting' is entirely dependent on your point of view. Are you looking for copy protection? How do you suspect it is done. When in the program execution does it show up? Are you looking to do a security audit of the program? Is there any sloppy string usage? Which functions use strcmp, sprintf, etc? Which use malloc? Is there a possibility of improper memory allocation?

Finding key functions

If we can narrow down our search to just a few functions that are relevant to our objective, our lives should be much easier.

Finding main()

Regardless of our objective, it is almost always helpful to know where main() lies. Unfortunately, when debugging symbols are removed, this is not always easy.

In Linux, program execution actually begins at the location defined by the _start symbol, which is provided by gcc in the crt0 libraries (check gcc -v for location). Execution then continues to __libc_start_main(), which calls _init() for each library in the program space. Each _init() then calls any global constructors you may have in that particular library. Global constructors can be created by making global instances of C++ classes with a constructor, or by specifying __attribute__((constructor)) after a function prototype. After this, execution is finally transferred to main.

The easiest technique is to try to use our friends ltrace and gdb together with our disassembled output. Checking the return address of the first few functions of ltrace -i, and cross referencing that to our assembly output and function call tree should give us a pretty good idea where main is. We may have to try to trick the program into exiting early, or printout out an error message before it gets too deep into its call stack.

Other techniques exist. For example, we can LD_PRELOAD a .c file with a constructor function in it. We can then set a breakpoint to a libc function that it calls that is also in the main executable, and `finish` and `stepi` until we are satisfied that we have found main.

Even better, we could just set a breakpoint in the function __libc_start_main (which is a libc function, and thus we will always have a symbol for it), and do the same technique of finishing and stepping until we reach what looks like main to us.

At worst, even without a frame pointer, we should be able to get the address of a function early enough in the execution chain for us to consider it to be main.

Finding other interesting functions

Its probably a good idea to make a list of all functions that call exit. These may be of use to us. Other techniques for tracking down interesting functions include:

1. Checking for which functions call obscure gui construction widgets used in a dialog box asking for a product serial number
2. Checking the string references to find out which functions reference strings that we are interested in. For example, if a program outputs the text "Already registered." knowing what function outputs this string is helpful in figuring out the protection this particular program uses.
3. Running a program in gdb, then hitting control C when it begins to perform some interesting operation. using stepi N should slow things down and allow you to be more accurate. Sometimes this is too slow however. Find a commonly called function, set a breakpoint, and try doing cont N.
4. Checking which functions call functions in the BSD socket layer

Plotting out program flow

Plot out execution paths into a tree from main, especially to your function(s) of interest. You can use disasm.pl to generate call graphs with the --graph option. Using it enables the script to generate file called call_graph. It contains definition of the call graph in a format used by a popular graphing tool called dot. Feeding this definition file in dot will give you a nice (probably pretty huge) graphics file with visual representation of the call graph. It is pretty amazing. Definitely try it with some small program.

Further analysis will have to hold off until we understand some assembly.

Prev | Up | Next
Chapter 4. Determining Program Behavior | Home | Chapter 6. Understanding Assembly

Chapter 6. Understanding Assembly

Table of Contents

Since the output of all of these tools is in AT&T syntax, those of you who know Intel/MASM syntax have a bit of re-learning to do.

Assembly language is one step closer to the hardware than high level languages like C and C++. So to understand assembly, you have to understand how the hardware works. Lets start with a set of memory locations known as the CPU registers.

Registers

Registers are like the local variables of the CPU, except there are a fixed number of them. For the ix86 CPU, there are only 4 main registers for doing integer calculations: A, B, C, and D. Each of these 4 registers can be accessed 4 different ways: as a 32 bit value (%eax), as a 16 bit value (%ax), and as a low and a high 8 bit value (%al and %ah). There are five more registers that you will see used occasionally - namely SI, DI, SP and BP. SI and DI are around from the DOS days when people used 64k segmented addressing, and as it turns out, may be used as integer like normal registers now. SP and BP are two special registers used to handle an area of memory called the stack. There is one last register, the instruction pointer IP that you may not modify directly, but is changed through jmps and calls. Its value is the address of the next instruction to execute. (FIXME: Check this)

Note: If gcc was called with the -fomit-frame-pointer, the BP register is freed up to be used as an extra integer register.

The stack

What is A stack?

A stack is what is called a Last In, First Out data structure or LIFO. Think of it as a stack of plates. The most recent (last) plate pushed on top of the stack is the first one to be removed. This allows us to manage the stack with only one register if need be, namely the stack pointer or SP register.

What is THE stack?

The stack is a region of memory that is present throughout the entire lifetime of a program. It is where local variables are stored, and it is also how function call arguments are passed.

On all modern computers, the stack is said to grow down, that is, as elements are pushed on to it, the SP register is decremented by the size of the element pushed. From our earlier analogy, its as if the stack of plates where hung from the ceiling, new plates were inserted at the bottom, and the whole stack some sort of catch to stop them all from dumping out. That catch would be the SP register.

So the stack starts from a high memory address, and works down to a lower address. This is because another section of memory called the heap grows up, and its handy to have the two of them grow towards eachother to fill in a single empty hole in the program address space.

Note: It is easy to become confused when dealing with the stack. Remember that while it may grow down, variables are still addressed sequentially upwards. So an array of char b[4] at esp of 80 will have b[0] at 80 (right at the stack pointer), b[1] above that at 81, b[2] at 82, and b[3] at 83, which is where the stack pointer was before the push. The next push will then place the stack pointer at 76.

Working with the stack

There are two instructions that deal with the stack directly: push and pop. Each take a register or value as an argument. Push will place its argument onto the stack, and then decrement the SP by the size of its argument (4 for pushl, 2 for pushw, 1 for pushb). //FIXME (What is pushl and push b) Pop copies the value on the top of the stack into its argument, then increments SP. Pusha and popa push and pop all the registers with one instruction. Because of speed considerations, the value is not touched, just the SP register is changed to point to the next location ot the stack. So SP is always pointing to the top value of the stack and not at invalid memory.

Normal arithmetic expressions can also be used to modify SP to make space for working directly with stack memory with other instructions.

How gcc works with the stack

Right before a function is called, its arguments are pushed onto the stack in reverse order. Then the call instruction pushes the address of the next instruction (ie the value of IP after call) onto the stack, and then the CPU begins executing the address of the call by copying that value into the invisible instruction pointer (IP) register.

The called function then starts with what is known as the function prolog, which pushes the current base pointer onto the stack, and then copies the current stack pointer to the base pointer, and then subtracts from SP enough space to hold all local variables (and then some!). The base pointer is then used to reference variables and parameters during function execution, since its value is not affected by pushes and pops. Thus, parameters all have fixed positive offsets from the BP, where as local variables all have fixed negative offsets from the BP.

At the end of function execution, the base pointer is copied to the stack pointer during ret, and the return address is popped off the stack and placed into the invisible IP register to return to the caller function.

Note: Unless -fomit-frame-pointer is specified, gcc always generates code that references local variables by negative offsets from the BP instead of positive offsets from the SP.

Two's complement

What is it?

Two's complement is specific way signed integers are represented in pretty much all modern computers. This is due to the fact that two's complement form has several advantages:

1. The same rules for addition apply, no extra work is required to compute the sum of negative integers.
2. Easy to negate a number.
3. The most significant bit tells you the sign: 0 is positive, 1 is negative.

It should be noted that when using signed values the ranges of number that can be represented by a specific number of bits is less than the usual. The range is -2^{n-1} to +2^{n-1}-1

Conversion

There are several ways to convert any unsigned binary number into signed two's complement form. The most intuitive and easy to remember is the following Complement each bit of the number and add one. Let's find how -13 is represented, so we convert it into its binary form:

```
0000 1101

Then invert all the bits.
1111 0010

Now add one to it.
1111 0011

So 1111 0011 is -13 in two's complement.
```

Second method is to complement all the bits to the left of the rightmost 1 bit, but not including it (but not the rightmost bit, for example 0001 0100). It sounds a bit complicated, but is easier once you figure out how it is done. Let's get back to the example of -13.

```
0000 1101
        ^
Invert the bits to the left of the rightmost one.
1111 0011
```

There you go. We get the number without second step of adding one. It can be proven why this method works, but we are not in class. Yet a third method is to subtract the number from 2^n. Here is how it works.

```
 1000 0000
```

```
-
 0000 1101
 ---------
 1111 0011
```

There may be other ways of doing it, but if you master those, you will not need to remember any more. To convert a negative number in two's complement, you apply the exact same procedure as described and you get back the positive value for the number.

From reverse engineering angle

Now that we know what two's complement is let's look at some examples of this type of representation in reverse engineering process. Using one of the tools discussed earlier, objdump and the wrapper disasm.pl, let's look at the ls command binary. If you look at function7 (which starts at address 80495a8), lines like the following appear frequently:

```
 80495be:       83 c4 f8                add    $0xfffffff8,%esp
```

What does this instruction do? It just adds some constant to the stack pointer register (%esp). There are two ways you can look at this constant. It is either a huge unsigned number or two's complement negative number. Since we just add to the stack pointer, it does not make sense to be big number, so let's find what is the value of this number.

```
  f    f    f    f    f    f    f    8
1111 1111 1111 1111 1111 1111 1111 1000

0000 0000 0000 0000 0000 0000 0000 1000
  0    0    0    0    0    0    0    8
```

Now we can see that this is just the negative of 0x00000008 or just plain -8 in decimal. If you think about this, what this line does is decrement the stack pointer by 8 bytes (allocate more space).

Byte Ordering

Why this section? One simple reason - different platforms use different byte ordering. There are two different orderings - little endian and big endian. Some of you are may be what byte ordering actually is? Byte ordering refers to the physical layout of data in memory. When a data structure or data type is represented by more than one byte, the ordering of bytes matter. For example if we consider a long (4 bytes) let's label the least significant byte 0 and the most significant one 3. If we are on little endian machine the long will be represented in memory like this (yeah, some machines do not allow addressable bytes, but let's forget about this): 0x040 0 0x041 1 0x042 2 0x043 3 On a big endian machine on the other hand, the long will be layed out like that: 0x040 3 0x041 2 0x042 1 0x043 0 Now let's look at an example. The easiest way to see the difference in byte ordering is to look at how string is stored in memory on different architectures. Here is an example program that will demonstrate it.

```
#include <stdio.h>
```

```
int main() {

        char* test = "this is a string";

        printf("%s\n", test);
}
```

We compiled it and here is the output of two different ways of disassembling it first on Solaris machine (Linux xxxxxx 2.4.16 #1 Tue Dec 11 01:57:19 EST 2001 sparc64 unknown): objdump

```
     11850:       74 68 69 73     call  d1a2be1c <_end+0xd1a0a394>
     11854:       20 69 73 20     unknown
     11858:       61 20 73 74     call  8482e628 <_end+0x8480cba0>
     1185c:       72 69 6e 67     call  c9a6d1f8 <_end+0xc9a4b770>
     11860:       00 00 00 00     unimp  0
```

gdb

```
     0x11850 <_IO_stdin_used+8>:     0x74686973      0x20697320      0x61207374
0x72696e67
```

Now let's look at how the memory itself is organized and how the string is represented:

```
Address          Code    Letter
--------------------------
0x11850          74      t
0x11851          68      h
0x11852          69      i
0x11853          73      s

0x11854          20
0x11855          69      i
0x11856          73      s
0x11857          20

0x11858          61      a
0x11859          20
0x1185a          73      s
0x1185b          74      t

0x1185c          72      r
0x1185d          69      i
0x1185e          6e      n
0x1185f          67      g

0x11860          00
```

And if we do the same on Intel machine (Linux xxxxxx 2.4.17 #17 Thu Jan 31 23:34:35 CST 2002 i686 unknown) this is what we get:

```
Address         Code    Letter
--------------------------
0x8048420       73         s
0x8048421       69         i
0x8048422       68         h
0x8048423       74         t

0x8048424       20
0x8048425       73         s
0x8048426       69         i
0x8048427       20

0x8048428       74         t
0x8048429       73         s
0x804842a       20
0x804842b       61         a

0x804842c       67         g
0x804842d       6e         n
0x804842e       69         i
0x804842f       72         r
```

At first glance of the x86 architecture you may miss that this actually is the string we are looking for. This is the difference in byte ordering. In order for different hosts on the same network to be able to communicate and the exchanged data to make sense, they agree on common byte ordering. In modern networking the data is transmitted in big endian byte ordering i.e. most significant byte comes first. On the i80x86 the host byte order is Least Significant Byte first, whereas the network byte order, as used on the Internet, is Most Significant Byte first.

Reading Assembly

Keep track of the stack and registers

The secret to understanding assembly code is to always work with a sheet of paper and a pencil. When you first sit down, draw out a table for all 6 registers A, B, C, D, SI, and DI. Keep track of the high and low portions as well. Each new line of this table should represent a modification of a register, so the last value in each register column is the current value of that register.

Next, draw out a long column for the stack, and leave space on the sides to place the BP and SP registers as they move down. Be sure to write all values into the stack as they are placed there, including ret and the stored BP.

AT&T syntax

In AT&T syntax, all instructions are of the form:

mnemonic src, dest

Standalone numerical constants are prepended with a $. Hexadecimal numbers always start with 0x (as opposed to ending in h). Registers are specified with a % sign, ie %eax.

Dereferencing or pointer representation is of the form disp(%base, %index, scale), where the resulting address is disp + %base + %index*scale. disp and scale are constants (no $), and %base and %index are registers. Any of these 4 may be omitted, leaving either blank space and then a comma, or simply leaving off the argument, and all remaining arguments. For example, 4(%eax) means memory address 4+%eax, where as (,%eax,4) means %eax*4. This compact notation makes array indexing easy.

Intel Instruction Set

From here, it is simply a matter of understanding what each assembly mnemonic does. Most common mnemonics are obvious, but you can find a complete description of all the Intel instructions (in agonizing detail) at Intel's Developer Site. Volume 2 contains the instruction list. Keep in mind that in Intel syntax, operands are in the reverse order of AT&T syntax (ie, mnemonic dest,src).

Know Your Compiler

In order to learn to read assembly effectively, you really have to know what type of code your compiler likes to generate in certain situations. If you learn to recognize what a while loop, a for loop, an if-else statement all look like in assembly, you can learn to get a general feel for code more quickly. There are also a few tricks that GCC performs that may seem unintuitive at first to the neophyte reverse engineer, even if they already know how to forward-engineer in assembly.

Basic Control Structures

In assembly, the only flow control mechanisms are branching and calling. So every control structure is built up from a combination of goto's and conditional branches. Lets look at some specific examples.

Function Calls

So we've mentioned that function calls use the stack to pass arguments. But where does that leave return values? And what about local variables?

Local variables are also on the stack, just below the base pointer instead of above. But if you thought that a return value was a pop off of the stack, you were wrong! GCC places the return value of a particular function into the eax register at the end of that function. Upon calling a function with a return value, it knows to copy the eax register into whatever variable will store that return value.

So lets see some gcc output for function calls. Get your paper ready, we're going to need to draw our stack and register table to follow these. Yeah yeah, it seems like a hassle, and you're sure you can do without it. We know, we know. But humor us. If you at least practice the methodical way a few times, doing things in your head will become easier later.

Example .c file and gcc output with no optimization, with -O2, and with -O3 -fomit-frame-pointer To get the most out of these examples, start at main, and trace execution throughout the executable. Do the low optimization first, and then move up to higher levels. The comments assume you are progressing in that order. FIXME: We may want to split these out into several simpler example files, to avoid overwhelming people all at once.

The if statement

The if statement is represented in assembly as a test followed by a jump. The thing to notice is that sometimes the body of the if statement is what is jumped to, as opposed to being jumped over as your C code may specify. This means that the condition for the jump will often be the negation of the condition for your if statement.

Example .c file and gcc output with no optimization, with -O2, and with -O3 -fomit-frame-pointer

The if..else statement

So we've seen that if statements are usually done by doing a single jump over the statement body. If..else statements operate the same way, except with an unconditional jump at the end of the if statement body that diverts execution flow to the end of the else body.

Example .c file and gcc output with no optimization, with -O2, and with -O3 -fomit-frame-pointer

If..else..if statements

Adding another if in an else clause works the same way as having an if statement inside an else clause. We just simply jump to another label if it evaluates to false, and if the first if statement evaluates as true, at the bottom of it we simply jump past both the else if and any remaining else clauses.

Example .c file and gcc output with no optimization, with -O2, and with -O3 -fomit-frame-pointer

Complicated if statements

Of course, if statements can get much more complicated than the above examples. They can contain boolean short-circuits, function calls, nested-ifs, etc.

The while loop

Think about the while loop for a second. Think about how it operates. Basically, you could write a while loop with an if and a goto statement inside the if body to the top of the loop. So, since the only branching mechanisms we have in assembly are jumps and calls, while loops are just if statements with a jmp back to the top at the bottom.

Example .c file and gcc output with no optimization, with -O2, and with -O3 -fomit-frame-pointer

The for loop

So lets rewrite the above loop as a for loop, to see if our professors were lying to us when they said these loops were equivalent.

Example .c file and gcc output with no optimization, with -O2, and with -O3 -fomit-frame-pointer

The do...while loop

Do while loops are a bit different than for and while loops in that they allow execution of the loop body to occur at least once. As such, their comparison instructions take place at the bottom of the loop as opposed to the top. Observe:

Example .c file and gcc output with no optimization, with -O2, and with -O3 -fomit-frame-pointer

Arrays

Arrays on the stack

Arrays on the stack are just memory regions that we access with variations on the disp(%base, %index, scale) idea presented earlier. So lets start with a warm-up consisting of a simple char array where we let libc do all the work.

Example .c file and gcc output with no optimization, with -O2, and with -O3 -fomit-frame-pointer

So lets do another example where we do all the work. One dimensional arrays are the easiest, as they are simply a chunk of memory that is the number of elements times the size of each element.

Example .c file and gcc output with no optimization, with -O2, and with -O3 -fomit-frame-pointer

Two dimensional arrays are actually just an abstraction that makes working with memory easier in C. A 2D array on the stack is just one long 1D array that the C compiler divides for us to make it manageable. To parameterize things, an array declared as: type array[dim2][dim1]; is really a 1D array of length dim2*dim1*type. The C compiler handles array indexing as follows: array[i][j] is the memory location array + i*dim1*type + j*type. So it divides our 1D array into dim2 sections, each dim1*type long.

FIXME: Graphics to illustrate this.

Example .c file and gcc output with no optimization, with -O2, and with -O3 -fomit-frame-pointer

As I tell my introductory computer science students, the best way to think of higher dimensional arrays is to think of a set of arrays of the next lower dimension. So the best way to think about how a 3D array can be jammed into a 1D array is to think about how a set of 2D arrays would be jammed into a 1D array: one right after another. So for array declared as *type array[dim3][dim2][dim1];*, array[i][j][k] means array + i*dim2*dim1*type + j*dim1*type + k*type. So this means just by looking at the assembly multiplications of the indexing variables, we should be able to determine n-1 dimensions of any n dimensional array. The remaining dimension can be determined from the total size, or the bounds of some initialization loop.

FIXME: Diagram/graphics to show this

Example .c file and gcc output with no optimization, with -O2, and with -O3 -fomit-frame-pointer

Arrays through malloc

Structs

Using structs

Structures (structs) are a convenient way of managing related variables without having to write a class to encapsulate all of them. A structure is essentially a class without any member functions. Structures are used VERY often in C in order to avoid passing several variables back and forth between functions. Instead of passing all the variables, a common practice is to encapsulate all of them in a struct and just pass the location of the struct in memory to the function that needs access to those variables. Structures in C++ are declared like this:

```
struct a
{
   int first;
   float second;
   char *third;
};
```

Don't forget that ; after the last brace. Structs can store any type of variable that you would normally be able to declare anywhere in your program. To access a variable in a struct you use the dot (.) operator. For example, to assign 5 to the variable first in the struct a, do

```
a.first = 5;
```

Arrays of structs

Arrays of structs are created just as you would create an array of any other variable. Using the declaration of a above, an array of a structs of size 10 would be declared like this:

```
struct a stuctarray[10];
```

Note the use of the struct keyword, followed by the name of the struct declared, followed by the name of the array.

The code above declares a static array of structs. This means that space will be allocated for this array during load time (FIXME: Check this). Struct arrays can also be declared as pointers so that space for individual elements can be allocated at run time as it is needed. (FIXME: Um...how is this done?...time to brush up on C).

Passing structs

Returning structs

GCC handles structs a bit oddly. When you have a function that returns a struct, what gcc does is actually push the address of the struct onto the stack just before calling the function (as if the first argument to the function was a pointer to the struct that will contain the return i value). Then, inside the function, code is generated to modify the struct through this address. At the end of the function, the value of %eax contains a pointer to the struct that was passed on to the stack. So instead of the normal convention of having %eax store the return value, %eax stores a pointer to the return value, and the return value is modified directly inside of the function.

Example .c file and gcc output with no optimization, with -O2, and with -O3 -fomit-frame-pointer

Classes (ie C++ code)

C with Classes

Inheritance

Virtual functions

Operator Overloading

Templates

Global variables

Exercises

These examples were all compiled using GCC 2.95.4 under Debian 3.0/Testing. A good exercise would be to go compile some of these examples with GCC 3.0 under high optimizations, changing some things around and viewing the resulting asm to get a feel for that new compiler and how it does things, as code it generates will begin to become more ubiquitous as time goes on. It was still considered rather unstable as of this writing, so we opted for the older GCC for all these examples for that reason.

Writing Inline Assembly

Calling Conventions

Prev Up Next

Chapter 5. Determining Interesting Functions Home Chapter 7. Debugging

Chapter 7. Debugging

Table of Contents

User-level Debugging

gdb

gdb is the GNU debugger. It is very intimidating to most people, but there really is no reason for it to be. It is very well done for a command line debugger. There is a nice GUI front end to it known as DDD, but our purposes will require a closer relationship with the command line.

gdb has a nice built-in help system organized by topic. typing help will show you the categories. The main commands we will be interested in are run, break, cont, stepi, finish, disassemble, bt, info [registers/frame], and x. Every command in gdb can be followed by a number N, which means repeat N times. For example, stepi 1000 will step over 1000 assembly instructions.

-> Example using gdb to set breakpoints in functions with and without debugging symbols.

-> FIXME: Test watchpoints

Using windbg

WinDbg is part of the standart Debugging Tools for Windows that everyone can download for free from. Microsoft offers few different debuggers, which use common commands for most operations and ofcourse there are cases where they differ. Since WinDbg is a GUI program, all operations are supposed to be done using the provided visual components. There is also a command line embeded in the debugger, which lets you type commands just like if you were to use a console debugger like **ntsd**.

Breakpoints

Breakpoints can be set, unset, or listed with the GUI by using Edit->Breakpoints or the shortcut keys Alt+F9. From the command line one can set breakpoints using the **bp** command, list them using **bl** command, and delete them using **bc** command. One can set breakpoints both on function names (provided the symbol files are available) or on a memory address.

Stack operations

Reading and Writing to Memory

Tips and tricks

Using softice

Kernel-level Debugging

Kernel-level debugging is useful if you want to attempt to figure out how a particular device driver is working, or if you want more information on a particular kernel entry point/API. Unfortunately, the support for kernel debugging is much better under Windows than it is under Linux. Fortunately, under Linux we have the source :)

Using kd

Using softice

Using gdb

Using the kernel profiling/hacking option

Prev Up Next
Chapter 6. Understanding Assembly Home Chapter 8. Executable formats

Chapter 8. Executable formats

Table of Contents

Working with the ELF Program Format

So at this point we now know how to write our programs on an extremely low level, and thus produce an executable file that very closely matches what we want. But the question is, how is our program code now actually stored on disk?

Well, recall that when a program runs, we start at the _start function, and move on from there to __libc_start_main, and eventually to main, which is our code. So somehow the operating system is gathering together a whole lot of code from various places, and loading it into memory and then running it. How does it know what code goes where?

The answer on Linux and UNIX is the ELF binary specification. ELF specifies a standard format for mapping your code on disk to a complete executable image in memory that consists of your code, a stack, a heap (for malloc), and all the libraries you link against.

So lets provide an overview of the information needed for our purposes here, and refer the user to the ELF spec to fill in the details if they wish. We'll start from the beginning of a typical executable and work our way down.

ELF Layout

There are three header areas in an ELF file: The main ELF file header, the program headers, and then the section headers. The program code lies in between the program headers and the section headers.

TODO: Insert figure here to show a typical ELF layout.

NOTE: ELF is extremely flexible. Many of these sections can be shunk, expanded, removed, etc. In fact, it is not outside the realm of possibility that some programs may deliberately make abnormal, yet valid ELF headers and files to try to make reverse engineering difficult (vmware does this, for example).

The Main ELF File Header

The main elf header basically tells us where everything is located in the file. It comes at the very beginning of the executable, and can be read directly from the first e_ehsize (default: 52) bytes of the file into this structure.

```
/* ELF File Header */
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;
```

The fields of interest to us are e_entry, e_phoff, e_shoff, and the sizes given. e_entry specifies the location of _start, e_phoff shows us where the array of program headers lies in relation to the start of the executable, and e_shoff shows us the same for the section headers.

The Program Headers

The next portion of the program are the ELF program headers. These describe the sections of the program that contain executable program code to get mapped into the program address space as it loads.

```
/* Program segment header.  */

typedef struct
{
  Elf32_Word    p_type;                 /* Segment type */
  Elf32_Off     p_offset;               /* Segment file offset */
  Elf32_Addr    p_vaddr;                /* Segment virtual address */
  Elf32_Addr    p_paddr;                /* Segment physical address */
  Elf32_Word    p_filesz;               /* Segment size in file */
  Elf32_Word    p_memsz;                /* Segment size in memory */
  Elf32_Word    p_flags;                /* Segment flags */
  Elf32_Word    p_align;                /* Segment alignment */
} Elf32_Phdr;
```

Keep in mind that there are going to a few of these (usually 2) end-to-end (ie forming an array of structs) in a typical ELF executable. The interesting fields in this structure are p_offset, p_filesz, and p_memsz, all of which we will need to make use of in the code modification chapter.

The ELF Body

The meat of the ELF file comes next. The actual locations and sizes of portions of the body are described by the program headers above, and contain the executable instructions from our assembly file, as well as string constants and global variable declarations. This will become important in the next chapter, program modification. (TODO: How to link to other chapters)

ELF Section Headers

The ELF section headers describe various named sections in an executable file. Each section has an entry in the section headers array, which is found at the bottom of the executable and has the following format:

```
/* Section header.  */

typedef struct
{
  Elf32_Word    sh_name;                /* Section name (string tbl index) */
  Elf32_Word    sh_type;                /* Section type */
  Elf32_Word    sh_flags;               /* Section flags */
  Elf32_Addr    sh_addr;                /* Section virtual addr at execution */
  Elf32_Off     sh_offset;              /* Section file offset */
  Elf32_Word    sh_size;                /* Section size in bytes */
  Elf32_Word    sh_link;                /* Link to another section */
  Elf32_Word    sh_info;                /* Additional section information */
  Elf32_Word    sh_addralign;           /* Section alignment */
  Elf32_Word    sh_entsize;             /* Entry size if section holds table */
} Elf32_Shdr;
```

The section headers are entirely optional, however. A list of common sections can be found on page 20 of the ELF Spec PDF

Editing ELF

Editing ELF is often desired during reverse engineering, especially when we want to insert bodies of code, or if we want to reverse engineer binaries with deliberately corrupted ELF headers.

Now you could edit these headers by hand using the <elf.h> header file and those above structures, but luckily there is already a nice editor called HT Editor that allows you to examine and modify all sections of an ELF program, from ELF header to actual instructions. (TODO: instructions, screenshots of HTE)

Do note that changing the size of various program sections in the ELF headers will most likely break things. We will get into how to edit ELF in more detail when we are talking about actual code insertion, which is the next chapter.

Working with the PE Program Format

Prev Up Next

Chapter 7. Debugging Home Chapter 9. Understanding Copy Protection

Chapter 9. Understanding Copy Protection

TODO: Not sure where to put this (perhaps in the intro? Different goals of reverse engineering? or perhaps as a part of the next section?) In any case, it should describe common methods to copy protection, and how it basically boils down to a conditional check in your program (with possible a little decryption). Basically it comes down to choosing between presenting techniques and then discussing how to use them, or first discussing how we can us the techniques we are about to discuss.. Which is better?

Chapter 10. Code Modification

Table of Contents

So now we know the tools to analyze our programs and find functions of interest to us even in programs without source code. We can understand the assembly that makes them up, and can write assembly of our own to do what we want. We know how a program looks on the disk and how that corresponds to what the program looks like in memory. Knowledge is power, and we know a lot. TODO: Read this: http://hcunix.org/hcunix/terran.txt

Reasons for Code Modification

Code modification is most useful if we wish to change the behavior of closed-source programs written by unenlightened authors. It is also handy when trying to skirt copy protection of various kinds.

Library Hooking

LD_PRELOAD

This is an environment variable that allows us to add a library to the execution of a particular program. Any functions in this library automatically override standard library functions. Sorry, you can't use this with suid programs.

Example:

```
% gcc -o preload.so -shared preload.c -ldl
% LD_PRELOAD=preload.so ssh students.uiuc.edu
```

Instruction Modification

Since the smallest unit of code is the instruction, it follows that the simplest form of code modification is instruction modification. In instruction modification, we are looking to change some property of a specific instruction. Recall from the assembly section that each instruction has 2 parts: The mnemonic and the arguments. So our choices are limited.

The best way to modify instructions is through HT Editor, which was mentioned earlier in the ELF section. HTE has a hex editor mode where we can edit the hex value of an instruction and see the assembly updated in real time. (TODO: instructions, screenshots of HTE)

Editing the arguments

Editing the arguments of an assembly instruction is easy. Simply look at the hex value of the assembly instruction's argument, and see where it lies in the hex bytes for that instruction. HTE will allow you to overwrite these values with values of your own. (Be careful with byte ordering!). TODO: Example1.

Editing the Mnemonic

This is far more tricky.

Single Instruction Insertion

Single Function Insertion

Use unused space as found by disasm.pl (be careful about main)

Multiple Function Insertion

Trickery.. We're working on a util to modify ELF programs and insert functions. What about using MMAP?? (P.S. Can you unmap executable memory to modify it... if they are doing an MD5 of their executable)

Attacking copy protection

Lest I be accused of hiding in my ivory tower, lets look a concrete application of these ideas, and some techniques (:

Prev Up Next

Chapter 9. Understanding Copy Protection Home Chapter 11. Network Application Interception

Chapter 11. Network Application Interception

Table of Contents

DCOM/RPC/CORBA/.NET Interception, general traffic sniffing, web services?

General Network Data Capture

What to do if the Network Layer is Encrypted

DCOM/RPC/CORBA

.NET

Web Services

Chapter 12. Buffer Overflows

Table of Contents

Sometimes you don't have access to the program code.

Stack Overflows

1-Byte Overflows

Returning to Libc

Attacking Countermeasures

Heap Overflows

Attacking hard copy protection

Chapter 13. TODO (Contribute!)

Table of Contents

Things that need to get done to this document. Note, none of these things are going to be particularly easy. But then again, neither was writing up the rest of this tutorial.

More detail

More detail is needed in some places, especially in the area of widget interception. (describing the event loop and suggesting good breakpoint places for GTK, Qt, Win32 might be nice)

Add resources and links section for each chapter (where applicable)

Update disasm.pl

The simpler things to do to this script would be to clean up the FIXME's, and add options to it (such as --no-show-raw-insn) Also, making an attempt at dereferencing pointers based on some heuristic would be nice. Check out this perl disassembler for ideas (not too many ideas.. its output format sucks).

If anyone is feeling extremely hardcore and wants to help modify Steve and Nasko's perl script to make the output more intuitive, feel free. A directed graph would be fantastic, automatic determination of main would also be great (use graph theory on your directed graph). There is also a utility called ptrace that is part of the LDasm project. Interfacing it (or gdb) with disasm.pl script to set a break point for each function would be a heroic task as well (because this would be the equivalent of ltrace, except for ALL functions in a program, not just the libs).

Do this for windows

If any of the dual booters in the crowd want to create a similar document for windows and/or give a talk, submissions are encouraged. Do note that in the meantime, all of these utils exist for windows as well, thanks to the cygwin project. (LINK). They should work the same there.

Do this for protocols

Protocol reverse engineering is a bit different than software engineering, tho many of the tools are the same. A tutorial on "reverse engineering" network protocols and data formats would also be helpful.

Do this for hardware

If anyone wants to present tactics for reverse engineering device drivers or electronic equipment, submissions are also welcome.

Prev Up Next
Chapter 12. Buffer Overflows Home Chapter 14. Extra Resources

Chapter 14. Extra Resources

Table of Contents

ELF Binary Specification

1. The Official Spec
2. Also in PDF
3. More interesting description
4. From a Linux Programmer's Perspective

Other Resources and amusements

1. LDasm project. LDasm is at best a passable disassembly tool (disasm.pl is FAR more useful), but it does come with a utility called ptrace, which allows you to view which instructions of a program actually execute. You can also give ptrace a list of addresses (for example, the list of functions found by disasm.pl) and have it step through those to show you which ones actually execute in your program.
2. Creating Teensy Executables in Linux
3. Microsoft COFF format
4. Attacking FlexLM is an essay written in 1998 on attacking a specific form of hard copy protection. There are several other essays on that site, but most of them cover material that we cover above, but with specific example programs.

```
        .file   "functions.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d, %d, %d\n"
.text
        .align 4
.globl function3args
        .type    function3args,@function
function3args:
        /* This push saves the ebp, and in combination with the move is called
         * the function prolog. */
        pushl %ebp /* at (%ebp) on the stack */
        movl %esp,%ebp

        /* This subl is used to allocate space for any local variables. In
         * this case we have none, and we can see the fact that this
         * instruction is useless because no stack references are negative
         * offsets from the %ebp (visualize or draw the stack to see this).
         * I'm not sure why GCC does this. */
        subl $8,%esp
        /* (%esp) == -8(%ebp) */

        /* remember our comments. This instruction copies the last argument of
         * the function to %eax*/
        movl 16(%ebp),%eax

        /* push this value as the last argument to the printf call.
         * Note: This is why we have an %ebp register, because this push will
         * affect the %esp, not the %ebp, and our references to local
         * variables all remain the same still. */
        pushl %eax
        /* (%esp) == -12(%ebp) */

        /* Now access the second argument of the function, and push it */
        movl 12(%ebp),%eax
        pushl %eax
        /* (%esp) == -16(%ebp) */

        /* Access the first argument of the function. Remember that the
         * remaining two things below 8(%ebp) are the return address at
         * 4(%ebp) and the old value of %ebp, which is at (%ebp) */
        movl 8(%ebp),%eax
        pushl %eax
        /* (%esp) == -20(%ebp) */

        /* Push the string onto the stack */
        pushl $.LC0
        /* (%esp) == -24(%ebp) */
        call printf
        /* (%esp) == -24(%ebp) because the stack is reset fixed after a call */

        /* Again, "pop" all 16 bytes of arguments off the stack */
        addl $16,%esp

        /* (%esp) == -8(%ebp) */

.L2:
        /* Leave copies the value of %ebp into %esp, effectively popping all
         * extra local variables and junk off the stack. It then pops the top
         * value off the stack (which is the saved %ebp) and stores it in %ebp
         *
         * So it is basically the reverse of the function
         * prolog, and implicityly removes any local variables and junk that
         * GCC may have thrown on the stack. This is key, because GCC loves to
         * throw junk on the stack for no reason. It is all taken care of at
         * function exit because of this instruction */
        leave

        /* (%esp) == (%ebp) == (old %ebp) just after call */

        /* pops the return address saved on the stack into %eip, and thus
         * execution transfers to just after the call */
        ret
.Lfe1:
        .size    function3args,.Lfe1-function3args
        .align 4
.globl function3argsRet
        .type    function3argsRet,@function
function3argsRet:
        pushl %ebp
        movl %esp,%ebp

        /* Move the first argument to %edx */
        /* The first argument is at 8 above the ebp. Ie it as at the lowest
         * address of all arguments. The rest are at higher address */
        movl 8(%ebp),%edx

        /* multiply the second argument with %edx, store in %edx */
        imull 12(%ebp),%edx

        /* multiply the third argument with %edx, store in %edx */
        imull 16(%ebp),%edx

        /* Move %edx to %eax. %eax is the return value */
        movl %edx,%eax

        /* Alignment junk */
        jmp .L3
        .p2align 4,,7
.L3:
        leave
        ret
.Lfe2:
        .size    function3argsRet,.Lfe2-function3argsRet
        .align 4
.globl functionPtrArg
        .type    functionPtrArg,@function
functionPtrArg:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp

        /* move the third argument (the pointer) into eax */
        movl 16(%ebp),%eax

        /* derefrence it. Remember how I said that leal does not deref, but
         * mov does? */
        movl (%eax),%edx

        /* push the rest of the args, and call printf */
        pushl %edx
        movl 12(%ebp),%eax
        pushl %eax
        movl 8(%ebp),%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $16,%esp
.L4:
        leave
        ret
.Lfe3:
        .size    functionPtrArg,.Lfe3-functionPtrArg
        .align 4
.globl functionPtrRet
        .type    functionPtrRet,@function
functionPtrRet:
        pushl %ebp
        movl %esp,%ebp

        /* Put the first argument of our function */
        movl 8(%ebp),%eax
        movl %eax,%edx

        /* put the address made by 0 + %edx*4 into register %eax */
        leal 0(,%edx,4),%eax
        movl %eax,%edx

        /* Add the third argument of our function (the pointer) to the result */
        addl 16(%ebp),%edx

        /* Put the second arg into eax */
        movl 12(%ebp),%eax
        movl %eax,%ecx

        /* put the address 0 + %ecx*4 into %eax. */
        leal 0(,%ecx,4),%eax

        /* add %eax to %edx, store in %edx.
         * If you were keeping track of the registers like you should have been,
         * you should now realize that %edx contains pointer + second_arg*4 +
         * third_ard*4. In other words, we know pointer is an integer pointer
         * because the scale was 4 during all the pointer arithmetic */
        addl %eax,%edx

        /* Put the result into the return value register %eax */
        movl %edx,%eax
        jmp .L5
        .p2align 4,,7
.L5:
        leave
        ret
.Lfe4:
        .size    functionPtrRet,.Lfe4-functionPtrRet
        .align 4
.globl functionLocalVars
        .type    functionLocalVars,@function
functionLocalVars:
        pushl   %ebp
        movl    %esp, %ebp
        /* so this is enough space for 4 integer variables, but sometimes GCC
         * allocates more space than it needs, especially in recent versions.
         * Note in this case, we have only THREE variables in our function.
         * But we will actually get to see GCC use this magic local variable
         * in a bit. Most times we aren't so lucky. */
        subl    $16, %esp

        /* recall 12 from ebp is the second 4-byte function argument (note
         * that if this function had non-integer arguments, 12(%ebp) might be
         * like the 3rd or 5th argument. Just something to keep in mind) */
        movl    12(%ebp), %eax

        /* XOR the second function arg with the first function arg */
        xorl    8(%ebp), %eax

        /* Store it in the first local variable. So the first local variable
         * now contains arg1 ^ arg2. This update of a local variable should
         * clue you into the completetion of a C statement.
         * In this case, we have determined that the statement was
         * local1 = arg1 ^ arg2;
         */
        movl    %eax, -4(%ebp)



        /* put the first arg into %edx */
        movl    8(%ebp), %edx

        /* Take the address of the second function arg.. */
        leal    12(%ebp), %eax

        /* put it into what appears to be the fourth local variable (again,
         * it could be the the 9th, 17th, etc)
         *
         * HOWEVER, NOTE: We do NOT have 4 local variables in the
         * corresponding C code. GCC has created a temporary here to do the
         * calculation. This is further evidence of non-optimized code. */
        movl    %eax, -16(%ebp)

        /* check your sheet for %edx */
        movl    %edx, %eax

        /* Move the fourth local variable into %ecx. So, following your sheet,
         * %ecx now contains the address of the second function arg. */
        movl    -16(%ebp), %ecx

        /* FIXME: BUH? */
        cltd

        /* So here's an odd intruction. Basically, if you check the Intel
         * Instruction set reference, you see that idiv takes a single
         * argument of either a register %reg or an indirected register (ie a
         * register containing a memory location, (%reg)) and then divides
         * %eax by the value in %reg or at memory location (%reg). The result is
         * stored in %eax, and the remainder is in %edx.
         */

        /* Do: %eax = %eax/(%ecx); %edx = %eax MOD (%ecx);
         * so from your sheet, %eax = arg1/arg2; %edx = arg1 MOD arg2 */
        idivl   (%ecx)

        /* Move result to second local variable. So local2 = arg1 / arg2; */
        movl    %eax, -8(%ebp)



        /* Move first arg to %edx */
        movl    8(%ebp), %edx

        /* Put the address of the second arg into %eax */
        leal    12(%ebp), %eax

        /* Use that temporary variable again */
        movl    %eax, -16(%ebp)
        movl    %edx, %eax
        movl    -16(%ebp), %ecx

        cltd

        /* %eax = %eax/(%ecx); %edx = %eax MOD (%ecx);
         * So, %eax = arg1/arg2; %edx = arg1 MOD arg2;
         */
        idivl   (%ecx)

        /* Store %edx into third local variable. So local3 = arg1 MOD arg2; */
        movl    %edx, -12(%ebp)



        /* Put the local2 into %eax */
        movl    -8(%ebp), %eax

        /* %eax = local1 | %eax */
        orl     -4(%ebp), %eax

        /* local3 = local1 | local2 */
        movl    %eax, -12(%ebp)

        /* Put local2 into eax */
        movl    -12(%ebp), %eax

        /* %eax = local1 & local2 */
        andl    8(%ebp), %eax

        /* Junk instruction that says return %eax */
        movl    %eax, %eax
        leave
        ret
.Lfe5:
        .size    functionLocalVars,.Lfe5-functionLocalVars
        .align 4
.globl main
        .type    main,@function
main:
        /* save ebp */
        pushl %ebp

        /* move esp to ebp so we can access vars from ebp */
        movl %esp,%ebp

        /* allocate stack space.. Notice that gcc likes to allocate WAY more
         * space than it needs in some cases.. why this is, I don't know.
         * We really only need 4 bytes of space here for our int a, and a
         * quick scroll through the function shows that -4(%ebp) is the only
         * local variable we use */
        subl $24,%esp
#APP
        nop
#NO_APP
        /* So here we see that GCC pushes some mystery arg onto the stack,
	 * and then the three arguments in reverse order, followed by the call
	 * to function3args. Remember that the call instruction places the
	 * address of the next instruction onto the stack. So at the entrance
	 * to function3args, esp points to the return address, and we have 20
	 * bytes above the esp, including ret and the mystery argument.
	 *
	 * However, since we are working on source generated without
	 * -fomit-frame-pointer, there will be a push of the ebp, and then the
	 * esp will be copied to ebp, and variables will be referenced from the
	 * ebp.
	 */
	addl $-4,%esp		/* 20(%ebp) after prolog */
	pushl $3		/* 16(ebp) */
	pushl $2		/* 12(%ebp) */
	pushl $1		/* 8(%ebp) */
	call function3args	/* 4(%ebp) */

	/* Go to function3args and see the comments there to see these
	 * variables in action */

	/* This stack ajustment is the same as popping all 4 arguments off the
	 * stack, ie the 3 integers and the mystery arg. */
	addl $16,%esp
#APP
	nop
#NO_APP

	/* So this function is the same exact deal as the previous, except we
	 * have a return value. GCC uses the eax register to store the return
	 * value  of a function.
	 * A good excercise would be to follow the stack along yourself with
	 * a sheet of paper for this example. */
	addl $-4,%esp
	pushl $3
	pushl $2
	pushl $1
	call function3argsRet
	addl $16,%esp

	/* Junk instruction, unoptimized code */
	movl %eax,%eax

	/* Notice now that %eax is copied into the first local variable */
	movl %eax,-4(%ebp)
#APP
	nop
#NO_APP

	/* This function exists as an example of what happens when you have a
	 * pointer as an argument. */
	addl $-4,%esp

	/* the lea instruction loads the effective address of its first
	 * argument and places it in the second. In other words, it simply
	 * adds the offset to the register being indexed, and then moves that
	 * into the destination.
	 *
	 * It is easy to become confused with this instruction, because it
	 * actually does NOT derefrence the first arg, where as a mov does.
	 */

	/* Load the address of the first local variable into %eax */
```

```
        leal -4(%ebp),%eax

        /* push it. Thus the pointer is the third argument */
        pushl %eax
        pushl $3
        pushl $1
        call functionPtrArg
        addl $16,%esp
#APP
        nop
#NO_APP
        /* The example is the same as the previous, except we return a
         * pointer */
        addl $-4,%esp

        leal -4(%ebp),%eax
        pushl %eax
        pushl $3
        pushl $1
        call functionPtrRet
        addl $16,%esp
        movl %eax,%eax
        /* Put the value in %eax into the second local variable. So the second
         * var must be an int pointer from out conclusions in functionPtrRet */
        movl %eax,-8(%ebp)
#APP
        nop
#NO_APP
        /* This example is intended to show how a function handles local
         * variables as always being negative offsets from the %ebp */

        /* Here we see another mystery stack allocation.. */
        subl    $8, %esp
        pushl   $2
        pushl   $1
        call    functionLocalVars
        addl    $16, %esp
        movl    %eax, %eax
        movl    %eax, -4(%ebp)
#APP
        nop
#NO_APP

.L6:
        leave
        ret
.Lfe6:
        .size    main,.Lfe6-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
// vim:noet
```

```
        .file   "functions.c"
        .version        "01.01"
gcc2_compiled.:
                .section        .rodata
.LC0:
        .string "%d, %d, %d\n"
.text
        .align 4
.globl function3args
        .type    function3args,@function
function3args:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        pushl   16(%ebp)
        pushl   12(%ebp)
        pushl   8(%ebp)
        pushl   $.LC0
        call    printf
        leave
        ret
.Lfe1:
        .size    function3args,.Lfe1-function3args
        .align 4
.globl function3argsRet
        .type    function3argsRet,@function
function3argsRet:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        imull   8(%ebp), %eax
        imull   16(%ebp), %eax
        popl    %ebp
        ret
.Lfe2:
        .size    function3argsRet,.Lfe2-function3argsRet
        .align 4
.globl functionPtrArg
        .type    functionPtrArg,@function
functionPtrArg:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    16(%ebp), %eax
        pushl   (%eax)
        pushl   12(%ebp)
        pushl   8(%ebp)
        pushl   $.LC0
        call    printf
        leave
        ret
.Lfe3:
        .size    functionPtrArg,.Lfe3-functionPtrArg
        .align 4
.globl functionPtrRet
        .type    functionPtrRet,@function
functionPtrRet:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        sall    $2, %eax
        addl    16(%ebp), %eax
        popl    %ebp
        ret
.Lfe4:
        .size    functionPtrRet,.Lfe4-functionPtrRet
        .align 4
.globl functionLocalVars
        .type    functionLocalVars,@function
functionLocalVars:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        pushl   %eax
        movl    8(%ebp), %ebx
        movl    %ebx, %eax
        movl    12(%ebp), %ecx
        cltd
        movl    %ebx, -8(%ebp)
        idivl   %ecx
        xorl    %ecx, -8(%ebp)
        movl    %eax, %ecx
        orl     %ecx, -8(%ebp)
        andl    -8(%ebp), %ebx
        movl    %ebx, %eax
        movl    -4(%ebp), %ebx
        leave
        ret
.Lfe5:
        .size    functionLocalVars,.Lfe5-functionLocalVars
        .align 4
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $8, %esp
#APP
        nop
#NO_APP
        pushl   $3
        pushl   $2
        pushl   $1
        call    function3args
#APP
        nop
#NO_APP
        addl    $12, %esp
        pushl   $3
        pushl   $2
        pushl   $1
        call    function3argsRet
        movl    %eax, -8(%ebp)
#APP
        nop
#NO_APP
        addl    $12, %esp
        leal    -8(%ebp), %ebx
        pushl   %ebx
        pushl   $3
        pushl   $1
        call    functionPtrArg
#APP
        nop
#NO_APP
        addl    $12, %esp
        pushl   %ebx
        pushl   $3
        pushl   $1
        call    functionPtrRet
#APP
        nop
#NO_APP
        popl    %edx
        popl    %ecx
        pushl   $2
        pushl   $1
        call    functionLocalVars
        movl    %eax, -8(%ebp)
#APP
        nop
#NO_APP
        movl    -4(%ebp), %ebx
        leave
        ret
.Lfe6:
        .size    main,.Lfe6-main
        .ident  "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.1 2.96-81)"
        .file   "functions.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d, %d, %d\n"
.text
        .align 4
.globl function3args
        .type    function3args,@function
function3args:
        subl $12,%esp
        pushl 24(%esp)
        pushl 24(%esp)
        pushl 24(%esp)
        pushl $.LC0
        call printf
        addl $16,%esp
        addl $12,%esp
        ret
.Lfe1:
        .size    function3args,.Lfe1-function3args
        .align 4
.globl function3argsRet
        .type    function3argsRet,@function
function3argsRet:
        movl 4(%esp),%eax
        imull 8(%esp),%eax
        imull 12(%esp),%eax
        ret
.Lfe2:
        .size    function3argsRet,.Lfe2-function3argsRet
        .align 4
.globl functionPtrArg
        .type    functionPtrArg,@function
functionPtrArg:
        subl $12,%esp
        movl 24(%esp),%eax
        pushl (%eax)
        pushl 24(%esp)
        pushl 24(%esp)
        pushl $.LC0
        call printf
        addl $16,%esp
        addl $12,%esp
        ret
.Lfe3:
        .size    functionPtrArg,.Lfe3-functionPtrArg
        .align 4
.globl functionPtrRet
        .type    functionPtrRet,@function
functionPtrRet:
        movl 4(%esp),%eax
        sall $2,%eax
        addl 12(%esp),%eax
        movl 8(%esp),%edx
        sall $2,%edx
        addl %edx,%eax
        ret
.Lfe4:
        .size    functionPtrRet,.Lfe4-functionPtrRet
        .align 4
.globl main
        .type    main,@function
main:
        subl $12,%esp
#APP
        nop
#NO_APP
        pushl $3
        pushl $2
        pushl $1
        pushl $.LC0
        call printf
        addl $16,%esp
#APP
        nop
        nop
#NO_APP
        pushl $6
        pushl $3
        pushl $1
        pushl $.LC0
        call printf
        addl $16,%esp
#APP
        nop
        nop
#NO_APP
        addl $12,%esp
        ret
.Lfe5:
        .size    main,.Lfe5-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "if.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "A is less than 0\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        /* save ebp */
        pushl %ebp

        /* move esp to ebp so we can access vars from ebp */
        movl %esp,%ebp

        /* allocate stack space */
        subl $24,%esp

        /* compare a to 0. The way this comparason works is that
         * the subtraction a - 0 is performed, and all of the flags on p65-66
         * of the Intel Basic Archetecture manual are updated. */
        cmpl $0,-4(%ebp)

        /* If you check the Intel Instruction Reference, the conditions for
         * jge are jump if SF == OF, ie jump if the result of the subtraction
         * was positive and there was no overflow, or jump if the
         * result of the subtraction was negative and there was an overflow */

        /* So the proper way to abstract all this away in your brain
         * is to think of cmp a,b and jXX as a pair that says:
         * "Jump if b XX a"
         */

        /* Jump if a ge 0, so jump to .L3 if (a >= 0) */
        jge .L3

        /* This code is now executed if (0 > a) */
        addl $-12,%esp
        pushl $.LC0
        call printf
        addl $16,%esp

.L3:
.L2:
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "if.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "A is less than 0\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        /* Save ebp */
        pushl %ebp
        /* Work off of sp */
        movl %esp,%ebp

        /* allocate space - Notice it goes unused. I'm still not sure why
         * gcc does this.
         */
        subl $8,%esp

        /*
         * Here we see that GCC has decided to use the test instruction in a
         * very wierd way. If you look at the Intel instruction reference
         * manual, you see that they are using the SF flag that is set with
         * the sign bit (remember the section we did on two's complement?)
         * of %eax AND %eax. This allows them to use jge, which
         * jumps on the condition that (SF = OF). Since OF is set to 0 by
         * test, the jge jumps to L18 on the condition that the sign bit of
         * %eax is 0. In otherwords, we jump to the end of the function
         * if ( %eax >= 0 ).
         *
         */
        testl %eax,%eax

        /* So the general way to abstract away a test a,a, jXX pair is to say:
         * "Jump if (a XX 0)"
         */

        /* if ( %eax >= 0) then jump */
        jge .L18

        /* following code is executed if (%eax < 0 ) */
        addl $-12,%esp
        pushl $.LC0
        call printf
.L18:
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "if.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "A is less than 0\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        /* Notice we have no function prolog with -fomit-frame-pointer */
        /* Also notice that we STILL allocate unneeded stack space.. go gcc! */
        subl $12,%esp

        /* Again that odd use of test */
        testl %eax,%eax

        /* jump if (%eax ge 0) */
        jge .L18


        addl $-12,%esp
        pushl $.LC0
        call printf
        addl $16,%esp
.L18:
        addl $12,%esp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "ifelse.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "A is less than 0\n"
        .align 32
.LC1:
        .string "A is greater than or equal to 0\n"
.LC2:
        .string "Leaving main\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        /* function prolog */
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp

        /* "Jump if -4(%ebp) ge 0" -> jump if (a >= 0) */
        cmpl $0,-4(%ebp)
        jge .L3

        /* This code executed if (a < 0) */
        addl $-12,%esp
        pushl $.LC0
        call printf
        addl $16,%esp

        /* Jump past the else clause to the unconditionally executed code */
        jmp .L4
        .p2align 4,,7
.L3:
        /* else { */
        addl $-12,%esp
        pushl $.LC1
        call printf
        addl $16,%esp
.L4:
        /* Unconditionally executed printf */
        addl $-12,%esp
        pushl $.LC2
        call printf
        addl $16,%esp
.L2:
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "ifelse.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "A is less than 0\n"
        .align 32
.LC1:
        .string "A is greater than or equal to 0\n"
.LC2:
        .string "Leaving main\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp

        /* jump if %eax ge 0 */
        testl %eax,%eax
        jge .L18

        /* code executed if (%eax < 0) */
        addl $-12,%esp
        pushl $.LC0

        /* Well now ain't this tricky. The printf call itself was determined
         * to be redunant since it was in both the if and the else clauses.
         * So it was moved right after the else section */


        /* Jump past else clause */
        jmp .L20
        .p2align 4,,7
.L18:
        /* Code executed if (%eax >= 0) */
        addl $-12,%esp
        pushl $.LC1
.L20:
        /* Factored-out shared printf call */
        call printf
        addl $16,%esp
        addl $-12,%esp
        pushl $.LC2
        call printf
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "ifelse.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "A is less than 0\n"
        .align 32
.LC1:
        .string "A is greater than or equal to 0\n"
.LC2:
        .string "Leaving main\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        subl $12,%esp
        /* not much in this file has changed as far as the if..else is
         * concerened */
        testl %eax,%eax
        jge .L18
        addl $-12,%esp
        pushl $.LC0
        jmp .L20
        .p2align 4,,7
.L18:
        addl $-12,%esp
        pushl $.LC1
.L20:
        call printf
        addl $16,%esp
        addl $-12,%esp
        pushl $.LC2
        call printf
        addl $16,%esp
        addl $12,%esp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "ifelseif.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "A is less than 0\n"
.LC1:
        .string "A is 0\n"
.LC2:
        .string "A > 0\n"
.LC3:
        .string "Leaving main\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp


        /* "Jump past if body if -4(%ebp) ge 0" */
        cmpl $0,-4(%ebp)
        jge .L3

        /* code executed if (a > 0) */
        addl $-12,%esp
        pushl $.LC0
        call printf
        addl $16,%esp

        /* jump past else if and else clause */
        jmp .L4
        .p2align 4,,7
.L3:
        /* else.. */
        /* jump past elseif body if -4(%ebp) ne 0 */
        cmpl $0,-4(%ebp)
        jne .L5

        /* code executed if (a == 0 */
        addl $-12,%esp
        pushl $.LC1
        call printf
        addl $16,%esp

        /* Jump past else */
        jmp .L4
        .p2align 4,,7
.L5:
        /* else */
        addl $-12,%esp
        pushl $.LC2
        call printf
        addl $16,%esp
.L6:
.L4:
        addl $-12,%esp
        pushl $.LC3
        call printf
        addl $16,%esp
```

```
.L2:
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "ifelseif.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "A is less than 0\n"
.LC1:
        .string "A is 0\n"
.LC2:
        .string "A > 0\n"
.LC3:
        .string "Leaving main\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp

        /* jump past if body if %eax ge 0 */
        testl %eax,%eax
        jge .L18

        addl $-12,%esp
        pushl $.LC0

        /* jump past elseif and else */
        jmp .L22
        .p2align 4,,7
.L18:
        /* jump if %eax ne 0 */
        testl %eax,%eax
        jne .L20

        addl $-12,%esp
        pushl $.LC1

        /* Jump past else */
        jmp .L22
        .p2align 4,,7
.L20:
        addl $-12,%esp
        pushl $.LC2
.L22:
        /* notice the factored printf again */
        call printf
        addl $16,%esp
        addl $-12,%esp
        pushl $.LC3
        call printf
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "ifelseif.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "A is less than 0\n"
.LC1:
        .string "A is 0\n"
.LC2:
        .string "A > 0\n"
.LC3:
        .string "Leaving main\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        /* again, not much has changed except this prolog. See if you can
         * follow this program's flow without help from the comments */
        subl $12,%esp
        testl %eax,%eax
        jge .L18
        addl $-12,%esp
        pushl $.LC0
        jmp .L22
        .p2align 4,,7
.L18:
        testl %eax,%eax
        jne .L20
        addl $-12,%esp
        pushl $.LC1
        jmp .L22
        .p2align 4,,7
.L20:
        addl $-12,%esp
        pushl $.LC2
.L22:
        call printf
        addl $16,%esp
        addl $-12,%esp
        pushl $.LC3
        call printf
        addl $16,%esp
        addl $12,%esp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "while.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp
        movl $0,-4(%ebp)
        .p2align 4,,7
.L3:
        cmpl $9,-4(%ebp)
        jle .L5
        jmp .L4
        .p2align 4,,7
.L5:
        addl $-8,%esp
        movl -4(%ebp),%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $16,%esp
        incl -4(%ebp)
        jmp .L3
        .p2align 4,,7
.L4:
.L2:
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "while.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $16,%esp
        pushl %esi
        pushl %ebx
        movl 12(%ebp),%esi
        xorl %ebx,%ebx
        jmp .L18
        .p2align 4,,7
.L20:
        addl $-8,%esp
        pushl %ebx
        pushl $.LC0
        call printf
        incl %ebx
        addl $16,%esp
.L18:
        addl $-12,%esp
        pushl 4(%esi)
        call atoi
        addl $16,%esp
        cmpl %eax,%ebx
        jl .L20
        leal -24(%ebp),%esp
        popl %ebx
        popl %esi
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "while.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        subl $24,%esp
        pushl %ebx
        xorl %ebx,%ebx
        .p2align 4,,7
.L20:
        addl $-8,%esp
        pushl %ebx
        pushl $.LC0
        call printf
        incl %ebx
        addl $16,%esp
        cmpl $9,%ebx
        jle .L20
        popl %ebx
        addl $24,%esp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "for.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp
        nop

        /* move 0 to var1 */
        movl $0,-4(%ebp)
        .p2align 4,,7
.L3:
        /* Jump if var1 le 9, ie if var1 <= 9 */
        cmpl $9,-4(%ebp)
        jle .L6
        /* exit loop */
        jmp .L4
        .p2align 4,,7
.L6:
        /* call to printf */
        addl $-8,%esp
        movl -4(%ebp),%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $16,%esp
.L5:
        /* var++ */
        incl -4(%ebp)
        jmp .L3
        .p2align 4,,7
        /* So we see that aside from some extra labels generated for each of
         * the sections of the loop, they are the same instructions */
.L4:
.L2:
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "for.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $16,%esp
        pushl %esi
        pushl %ebx
        movl 12(%ebp),%esi
        xorl %ebx,%ebx
        jmp .L18
        .p2align 4,,7
.L21:
        addl $-8,%esp
        pushl %ebx
        pushl $.LC0
        call printf
        addl $16,%esp
        incl %ebx
.L18:
        addl $-12,%esp
        pushl 4(%esi)
        call atoi
        addl $16,%esp
        cmpl %eax,%ebx
        jl .L21
        leal -24(%ebp),%esp
        popl %ebx
        popl %esi
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "for.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        subl $24,%esp
        pushl %ebx
        xorl %ebx,%ebx
        .p2align 4,,7
.L21:
        addl $-8,%esp
        pushl %ebx
        pushl $.LC0
        call printf
        addl $16,%esp
        incl %ebx
        cmpl $9,%ebx
        jle .L21
        popl %ebx
        addl $24,%esp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "dowhile.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp

        /* Move 0 to var1 */
        movl $0,-4(%ebp)
        .p2align 4,,7
.L3:
        /* call to printf */
        addl $-8,%esp
        movl -4(%ebp),%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $16,%esp

        /* var++ */
        incl -4(%ebp)
.L5:
        /* Now, here we see the comparason at the bottom, so that the loop
         * runs at least once before termination. Turns out the code for the
         * comarison is generated the exact same way */

        /* jump if var1 <= 9 */
        cmpl $9,-4(%ebp)
        jle .L6

        /* else quit */
        jmp .L4
        .p2align 4,,7
.L6:
        jmp .L3
        .p2align 4,,7
.L4:
.L2:
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "dowhile.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $16,%esp
        pushl %esi
        pushl %ebx
        movl 12(%ebp),%esi
        xorl %ebx,%ebx
        .p2align 4,,7
.L21:
        addl $-8,%esp
        pushl %ebx
        pushl $.LC0
        call printf
        incl %ebx
        addl $16,%esp
        addl $-12,%esp
        pushl 4(%esi)
        call atoi
        addl $16,%esp
        cmpl %eax,%ebx
        jl .L21
        leal -24(%ebp),%esp
        popl %ebx
        popl %esi
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "dowhile.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        subl $24,%esp
        pushl %ebx
        xorl %ebx,%ebx
        .p2align 4,,7
.L21:
        addl $-8,%esp
        pushl %ebx
        pushl $.LC0
        call printf
        incl %ebx
        addl $16,%esp
        cmpl $9,%ebx
        jle .L21
        popl %ebx
        addl $24,%esp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "array-stack-char.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "hello there, govna!"
.text
        .align 4
.globl charArray
        .type    charArray,@function
charArray:
        pushl %ebp
        movl %esp,%ebp
        /* Subtract enough space for the array and then some. Such large stack
         * allocations are a HUGE clue that somebody is working with arrays on
         * the stack. */
        subl $520,%esp


        /* mystery arg to strncpy */
        addl $-4,%esp

        /* This line is perplexing at first, but scan down. Its the length
         * argument to strncpy. This gives us the hint that GCC allocated 8
         * extra bytes on the stack */
        pushl $511
        /* string to copy */
        pushl $.LC0
        /* address of the buffer to copy into */
        leal -512(%ebp),%eax
        pushl %eax
        call strncpy
        /* Post-call stack adjust */
        addl $16,%esp

        /* more mystery args */
        addl $-12,%esp
        /* Strlen */
        pushl $.LC0
        call strlen
        /* stack ajust */
        addl $16,%esp

        /* Return value transfer (unoptimized) */
        movl %eax,%eax

        /* put address of string into edx */
        leal -512(%ebp),%edx



        movb $0,(%eax,%edx)
        /*
           Recall: disp(%base, %index, scale) = disp + %base + %index*scale.
           In this case, base and scale were omitted, so we have the address
           %eax + %edx. (Scale is assumed to be one). Since %eax contains the
           return value from strlen, we are doing string[strlen(.LC0)] = 0.
           In otherwords, we are null terminating the string, in case the
           strncpy call failed to copy everything. Think about this for a
           minute. This is a bug. Can you see why?

           Answer: If the strncpy call failed, LESS than .LC0 would have been
           copied because there wasn't enough room! Hence this is a bug that we            have
discovered through painstaking analysis of the assembly that the
           author of the C code overlooked! (To those of you who worry this may
           be a contrived example, I wrote the .c file, and didn't notice this
           bug until looking at the assembly just now).

           Techniques to use bugs like this to our advantage will be discussed
           later, in the buffer overflow chapter.
         */


        /* mystery arg */
        addl $-12,%esp
        leal -512(%ebp),%eax
        pushl %eax
        /*
           printArray is a bogus function that we call simply to prevent the
           optimizer from optimizing away all our code in future examples.
         */
        call printArray
        addl $16,%esp
.L2:
        leave
        ret
.Lfe1:
        .size    charArray,.Lfe1-charArray
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "array-stack-char.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "hello there, govna!"
.text
        .align 4
.globl charArray
        .type    charArray,@function
charArray:
        pushl %ebp
        movl %esp,%ebp
        subl $532,%esp
        pushl %ebx
        addl $-4,%esp
        pushl $511
        pushl $.LC0
        leal -512(%ebp),%ebx
        pushl %ebx
        call strncpy
        movb $0,-493(%ebp)
        addl $-12,%esp
        pushl %ebx
        call printArray
        movl -536(%ebp),%ebx
        leave
        ret
.Lfe1:
        .size    charArray,.Lfe1-charArray
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "array-stack-char.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "hello there, govna!"
.text
        .align 4
.globl charArray
        .type    charArray,@function
charArray:
        subl $536,%esp
        pushl %ebx
        addl $-4,%esp
        pushl $511
        pushl $.LC0
        leal 28(%esp),%ebx
        pushl %ebx
        call strncpy
        movb $0,51(%esp)
        addl $-12,%esp
        pushl %ebx
        call printArray
        addl $32,%esp
        popl %ebx
        addl $536,%esp
        ret
.Lfe1:
        .size    charArray,.Lfe1-charArray
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "array-stack-int1D.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl intArray
        .type    intArray,@function
intArray:
        pushl %ebp
        movl %esp,%ebp

        /* Woah thats a lot of space */
        subl $2072,%esp

        /* nop is a Null OPeration. It does nothing but padd our instruction
         * space */
        nop

        /* Set some variable var1 to zero. (Keep track of it on your stack
         * sheet!) */
        movl $0,-2052(%ebp)

        /* alignment noise */
        .p2align 4,,7
.L3:

        /* Scanning ahead, we see what looks like it could be a loop: Double
         * jump, label here, label after comparason.. */
        /* Recall: "Jump if -2052(%ebp) le $511" */
        cmpl $511,-2052(%ebp)
        jle .L7

        /* if var1 > 511, exit loop */
        jmp .L5
        .p2align 4,,7
.L6:

        /* put var1 in eax */
        movl -2052(%ebp),%eax
        movl %eax,%edx

        /* Here we see our indexing operation begin:
           Place var1*4 into %eax */
        leal 0(,%edx,4),%eax

        /* place the address of some nicely aligned quantity into %edx
         (A large array, perhaps?) */
        leal -2048(%ebp),%edx

        /* Place var1 into ecx */
        movl -2052(%ebp),%ecx

        /* *(%eax + %edx) = %ecx; -> array1[var1] = var1
           (because %eax = var1*4 */
        movl %ecx,(%eax,%edx)
.L5:
        /* var1++ */
        incl -2052(%ebp)
        /* loop */
        jmp .L4
        .p2align 4,,7
.L4:
        /* Printarray call to prevent over-optimization */
        addl $-12,%esp
        leal -2048(%ebp),%eax
        pushl %eax
        call printArray
        addl $16,%esp
.L2:
        leave
        ret
.Lfe1:
        .size    intArray,.Lfe1-intArray
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "array-stack-int1D.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl intArray
        .type    intArray,@function
intArray:
        pushl %ebp
        movl %esp,%ebp

        /* A whole lot of stack space is clue to an array */
        subl $2056,%esp

        /* leals are clue to the fact that we are going to be doing some more
         * indexing in the future. From this its save to assume that -2048
         * down from %ebp is our array, and local variables are after it. */
        leal -2048(%ebp),%edx

        movl $511,%ecx

        /* Here is the top of our array */
        leal -4(%ebp),%eax
        .p2align 4,,7
.L21:

        /* *%eax = %ecx;.. Note: 32bit integer operation */
        movl %ecx,(%eax)

        /* move %eax down by 4. We are now sure we're dealing with ints here */
        addl $-4,%eax

        /* Decrement counter */
        decl %ecx

        /* JNS means jump if not signed, ie if the result of the previous
         * instruction was not negative. So jump if %ecx >= 0 */
        jns .L21

        /* So can you predict the results of the following imaginary
         * printArray call? Our resulting code is a bit different than
         * the original code. Instead of running the loop forwards, the
         * optimizer has decided that we should start at index 511, and run
         * backwards until %ecx < 0. So the array is still numbered 0..511, we
         * just did it in reverse. Pretty strange optimization, eh?
         */

        addl $-12,%esp
        pushl %edx
        call printArray
        leave
        ret
.Lfe1:
        .size    intArray,.Lfe1-intArray
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
```

```
	.file	"array-stack-int1D.c"
	.version	"01.01"
gcc2_compiled.:
.text
	.align 4
.globl intArray
	.type	 intArray,@function
intArray:
	subl $2060,%esp
	movl %esp,%edx
	movl $511,%ecx
	leal 2044(%esp),%eax
	.p2align 4,,7
.L21:
	movl %ecx,(%eax)
	addl $-4,%eax
	decl %ecx
	jns .L21
	addl $-12,%esp
	pushl %edx
	call printArray
	addl $16,%esp
	addl $2060,%esp
	ret
.Lfe1:
	.size	 intArray,.Lfe1-intArray
	.ident	"GCC: (GNU) 2.95.4  (Debian prerelease)"
```

```
        .file   "array-stack-int2D.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl intArray2D
        .type    intArray2D,@function
intArray2D:
        pushl %ebp
        movl %esp,%ebp
        /* Lots of stack space.. Clue that we're working with arrays */
        subl $424,%esp
        nop

        /* Give -404(%ebp) the label var1 on your stack sheet, set it 0 */
        /* This also gives us a bound on the total array size.. Most likely
         * they specified the array first, then the vars */
        movl $0,-404(%ebp)
        .p2align 4,,7
.L3:
        /* Uh oh.. a loop! */
        /* "Jump if var1 le 9" -> Loop while var1 <= 9 */
        cmpl $9,-404(%ebp)
        jle .L6
        jmp .L4
        .p2align 4,,7
.L6:
        /* Lable this space var2 */
        movl $0,-408(%ebp)
        .p2align 4,,7
.L7:

        /* Hrmm.. could this be a nested loop? YEP! */

        /* "Loop while var2 <= 9" */
        cmpl $9,-408(%ebp)
        jle .L10
        jmp .L5
        .p2align 4,,7
.L10: /* Loop body */
        /* move var1 to eax */
        movl -404(%ebp),%eax
        /* Jump if var2 ne var1 */
        cmpl -408(%ebp),%eax
        jne .L11

        /* Code executed if (var2 == var1) */

        /* Put var2 into eax */
        movl -408(%ebp),%eax
        movl %eax,%edx

        /* Indexing operation coming! (%eax = var2*4*/
        leal 0(,%edx,4),%eax

        /* put var1 into ecx, then edx */
        movl -404(%ebp),%ecx
        movl %ecx,%edx

        /* The sal instruction bitshifts the operand left by the specified
         * number. It is basically a faster way of multiplying by powers of 2.*/
        /* %edx *= 4; (edx = var1*4)*/
        sall $2,%edx

        /* %edx = var1 * 5 */
        addl %ecx,%edx

        /* %ecx = var1 * 5 * 8 = var1 * 40 (hrmm.. 40 is 10*4... coincidence?)*/
        leal 0(,%edx,8),%ecx


        /* %eax = var1*40 + var2*4 */
        addl %ecx,%eax

        /* Put the base of the array into %edx */
        leal -400(%ebp),%edx

        /* put 1 into the address %eax + %edx. You see that gcc likes to use
         * the base and index backwards if there is no scale.. Lord only knows..
         *
         * The important thing to notice is that we have stored a 1 at memory
         * location array + var1*40 + var2*4, and we have done it HORRIBLY
         * inefficiently! (A human should have been able to do this with 2
         * leals and an add).
         *
         * Why 40 and 4? Well, recall that 2D arrays on the
         * stack of the form  'type array[dim2][dim1];'
         * are represented by a single array of size type*dim1*dim2. So
         * visualize long array as being divided into rows now (like text that
         * wraps around the screen). To get to the var1 row, we have to go past
         * var1*dim1*type cells, and to get to the var2 column, we have to add
         * on var2*type cells. Thus array[var1][var2] is
         *
         * array + var1*dim1*type + var2*type.
         */
        movl $1,(%eax,%edx)
        jmp .L9
        .p2align 4,,7
.L11:
        /* Else clause to if(var2 == var1) */

        /* put var2 into eax */
        movl -408(%ebp),%eax
        movl %eax,%edx

        /* eax now has var2*4 */
        leal 0(,%edx,4),%eax

        /* ecx has var1 */
        movl -404(%ebp),%ecx
        movl %ecx,%edx

        /* edx = var1*4 */
        sall $2,%edx
        /* edx = var1*5 (because ecx = var1) */
        addl %ecx,%edx

        /* ecx = var1*40 */
        leal 0(,%edx,8),%ecx

        /* eax = var1*40 + var2*4 */
        addl %ecx,%eax

        /* Base of our array in edx */
        leal -400(%ebp),%edx
        /* put the zero in eax */
        movl $0,(%eax,%edx)
.L12:
.L9:

        /* var2++ */
        incl -408(%ebp)
        jmp .L7
        .p2align 4,,7
.L8:
.L5:
        /* var1++ */
        incl -404(%ebp)
        jmp .L3
        .p2align 4,,7
.L4:

        /* So, can you visualize what this code is doing based on the assembly
         * we just went through without reverting back to the C code?
         * What does the 2D array look like after the program is done? Can you
         * draw it in 2D? How about in 1D? How about on the stack? (recall it
         * is on the stack) */


        /* Answer:
           So let's summarize:
           We have an outer loop that is iterating over var1 until it hits 10
            We have an inner loop that is iterating over var2 until it hits 10
             The inner loop sets array[var1][var2] to 1 if var1 == var2
             else it sets array[var1][var2] to 0.

           So can you draw the array now?
         */

        addl $-12,%esp
        leal -400(%ebp),%eax
        pushl %eax
        call printArray
        addl $16,%esp
.L2:
        leave
        ret
.Lfe1:
        .size    intArray2D,.Lfe1-intArray2D
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "array-stack-int2D.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl intArray2D
        .type    intArray2D,@function
intArray2D:
        pushl %ebp
        movl %esp,%ebp

        /* Huge allocation. Must be an array */
        subl $412,%esp

        /* preserve registers */
        pushl %edi
        pushl %esi
        pushl %ebx

        /* %ebx = 0 */
        xorl %ebx,%ebx

        /* Think about where -400(%ebp) is on the stack, and how it is
         * aligned. The fact that it is such a nice number really suggests
         * that we have the bottom of an array here
         * %eax = array */
        leal -400(%ebp),%eax

        /* So this is kind of odd.. the pointer to the bottom of array is
         * being stored on the stack. Just make a note of it and move on. */
        movl %eax,-404(%ebp)
        movl %eax,%edi
        .p2align 4,,7
.L21:
        /* %ecx = 0 */
        xorl %ecx,%ecx

        /* %edx = %ebx*4 */
        leal 0(,%ebx,4),%edx

        /* %esi = %ebx + 1 */
        leal 1(%ebx),%esi

        /* %eax = %ebx + %edx = %ebx*5 */
        leal (%ebx,%edx),%eax

        /* %eax = %eax*8 =  %ebx*40 */
        sall $3,%eax

        /* %edx = %ebx*40 + %ebx*4 */
        addl %eax,%edx

        /* %eax = %ebx*40 + array */
        addl %edi,%eax
        .p2align 4,,7
.L25:
        /* if(%ebx != %ecx) jump */
        cmpl %ecx,%ebx
        jne .L26

        /* code executed if(%ebx = %ecx) */

        /* array + %edx = 1 */
        movl $1,(%edx,%edi)
        jmp .L24
        .p2align 4,,7
.L26:
        movl $0,(%eax)
.L24:
        addl $4,%eax
        incl %ecx
        cmpl $9,%ecx
        jle .L25
        movl %esi,%ebx
        cmpl $9,%ebx
        jle .L21
        addl $-12,%esp
        movl -404(%ebp),%eax
        pushl %eax
        call printArray
        leal -424(%ebp),%esp
        popl %ebx
        popl %esi
        popl %edi
        leave
        ret
.Lfe1:
        .size    intArray2D,.Lfe1-intArray2D
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "array-stack-int2D.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl intArray2D
        .type    intArray2D,@function
intArray2D:
        subl $412,%esp
        pushl %ebp
        pushl %edi
        pushl %esi
        pushl %ebx
        xorl %ebx,%ebx
        leal 16(%esp),%ebp
        movl %ebp,%edi
        .p2align 4,,7
.L21:
        xorl %ecx,%ecx
        leal 0(,%ebx,4),%edx
        leal 1(%ebx),%esi
        leal (%ebx,%edx),%eax
        sall $3,%eax
        addl %eax,%edx
        addl %edi,%eax
        .p2align 4,,7
.L25:
        cmpl %ecx,%ebx
        jne .L26
        movl $1,(%edx,%edi)
        jmp .L24
        .p2align 4,,7
.L26:
        movl $0,(%eax)
.L24:
        addl $4,%eax
        incl %ecx
        cmpl $9,%ecx
        jle .L25
        movl %esi,%ebx
        cmpl $9,%ebx
        jle .L21
        addl $-12,%esp
        pushl %ebp
        call printArray
        addl $16,%esp
        popl %ebx
        popl %esi
        popl %edi
        popl %ebp
        addl $412,%esp
        ret
.Lfe1:
        .size    intArray2D,.Lfe1-intArray2D
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "array-stack-int3D.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl intArray3D
        .type    intArray3D,@function
intArray3D:
        pushl %ebp
        movl %esp,%ebp
        /* Woah thats a lot of memory */
        subl $1224,%esp
        nop
        /* Set var1 = 0 */
        movl $0,-1204(%ebp)
        .p2align 4,,7
.L3:
        /* While(var1 <= 2) */
        cmpl $2,-1204(%ebp)
        jle .L6
        jmp .L4
        .p2align 4,,7
.L6:
        /* set var2 = 0 */
        movl $0,-1208(%ebp)
        .p2align 4,,7
.L7:
        /* While(var2 <= 9) */
        cmpl $9,-1208(%ebp)
        jle .L10
        jmp .L5
        .p2align 4,,7
.L10:
        /* Set var3 = 0 */
        movl $0,-1212(%ebp)
        .p2align 4,,7
.L11:
        /* While(var3 <= 9) */
        cmpl $9,-1212(%ebp)
        jle .L14
        jmp .L9
        .p2align 4,,7
.L14:
        /* var2 -> eax */
        movl -1208(%ebp),%eax

        /* if(var2 != var3) then jump*/
        cmpl -1212(%ebp),%eax
        jne .L15

        /* code executed if(var2 == var3) */

        /* place var3 in eax */
        movl -1212(%ebp),%eax
        movl %eax,%edx

        /* eax = var3 *4 */
        leal 0(,%edx,4),%eax

        /* place var2 in ecx */
        movl -1208(%ebp),%ecx
```

```
	movl %ecx,%edx

	/* edx = var2*4 */
	sall $2,%edx
	/* edx = var2*5 */
	addl %ecx,%edx

	/* ecx = var2*40 */
	leal 0(,%edx,8),%ecx

	/* eax = var2*40 + var3 * 4 */
	addl %ecx,%eax

	/* ecx = var1 */
	movl -1204(%ebp),%ecx
	movl %ecx,%edx

	/* edx = var1*4 */
	sall $2,%edx
	/* edx = var1*5 */
	addl %ecx,%edx

	/* ecx = var1*20 */
	leal 0(,%edx,4),%ecx

	/* edx = var1*25 */
	addl %ecx,%edx
	movl %edx,%ecx

	/* ecx = var1*25*16 = var1*100*4 = var1*400 */
	sall $4,%ecx

	/* eax = var1*400 + var2*40 + var3*4 */
	addl %ecx,%eax

	/* edx = base of array */
	leal -1200(%ebp),%edx

	/* ecx = var1 */
	movl -1204(%ebp),%ecx

	/* set *(array + var1*400 + var2*40 + var3*4) = var1.
	 * So: array[var1][var2][var3] = var1;
	 *
	 * Can we guess the dimensions of our array at this point yet?
	 *
	 * From the formula given, 400 = dim2*dim1*type, 40 = dim1*type,
	 * 4=type.
	 *
	 * So type is int, dim1 is 10, dim2 is 10, dim3 is unknown.
	 * For a hint at dim3, what does the loop iterate var1 over?
	 * It executes so long as var1 <= 2. So our array is probably declared
	 * as:
	 * int array[3][10][10];
	 */
	movl %ecx,(%eax,%edx)
	jmp .L13
	.p2align 4,,7
.L15:
	/* else clause for if(var2 == var3) */

	/* this is pretty much the same code as above.. with one exception.. */
	movl -1212(%ebp),%eax
```

```
        movl %eax,%edx

        leal 0(,%edx,4),%eax
        movl -1208(%ebp),%ecx
        movl %ecx,%edx
        sall $2,%edx
        addl %ecx,%edx
        leal 0(,%edx,8),%ecx
        addl %ecx,%eax
        movl -1204(%ebp),%ecx
        movl %ecx,%edx
        sall $2,%edx
        addl %ecx,%edx
        leal 0(,%edx,4),%ecx
        addl %ecx,%edx
        movl %edx,%ecx
        sall $4,%ecx
        addl %ecx,%eax
        leal -1200(%ebp),%edx

        /* set *(array + var1*400 + var2*40 + var3*4) = 0 */
        movl $0,(%eax,%edx)
.L16:
.L13:
        /* var3++ */
        incl -1212(%ebp)
        jmp .L11
        .p2align 4,,7
.L12:
.L9:
        /* var2++ */
        incl -1208(%ebp)
        jmp .L7
        .p2align 4,,7
.L8:
.L5:
        /* var1++ */
        incl -1204(%ebp)
        jmp .L3
        .p2align 4,,7
.L4:

        /* So can you visualize what is going on with our 3D array?
         * What does it look like? You should be able to do this on your own
         * with little to no difficulty now.
         */

        addl $-12,%esp
        leal -1200(%ebp),%eax
        pushl %eax
        call printArray
        addl $16,%esp
.L2:
        leave
        ret
.Lfe1:
        .size    intArray3D,.Lfe1-intArray3D
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
        .file   "array-stack-int3D.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl intArray3D
        .type    intArray3D,@function
intArray3D:
        pushl %ebp
        movl %esp,%ebp
        subl $1228,%esp
        pushl %edi
        pushl %esi
        pushl %ebx
        movl $0,-1204(%ebp)
        leal -1200(%ebp),%eax
        movl %eax,-1212(%ebp)
        .p2align 4,,7
.L21:
        xorl %esi,%esi
        movl -1204(%ebp),%edx
        incl %edx
        movl %edx,-1208(%ebp)
        movl -1204(%ebp),%edi
        leal (%edi,%edi,4),%eax
        leal (%eax,%eax,4),%ebx
        sall $4,%ebx
        .p2align 4,,7
.L25:
        xorl %ecx,%ecx
        leal 0(,%esi,4),%edx
        leal 1(%esi),%eax
        movl %eax,-1216(%ebp)
        leal (%esi,%edx),%eax
        sall $3,%eax
        addl %eax,%edx
        addl %ebx,%edx
        addl %ebx,%eax
        .p2align 4,,7
.L29:
        cmpl %ecx,%esi
        jne .L30
        movl -1204(%ebp),%edi
        movl %edi,-1200(%edx,%ebp)
        jmp .L28
        .p2align 4,,7
.L30:
        movl $0,-1200(%eax,%ebp)
.L28:
        addl $4,%eax
        incl %ecx
        cmpl $9,%ecx
        jle .L29
        movl -1216(%ebp),%esi
        cmpl $9,%esi
        jle .L25
        movl -1208(%ebp),%eax
        movl %eax,-1204(%ebp)
        cmpl $2,%eax
        jle .L21
        addl $-12,%esp
```

```
	movl -1212(%ebp),%edx
	pushl %edx
	call printArray
	leal -1240(%ebp),%esp
	popl %ebx
	popl %esi
	popl %edi
	leave
	ret
.Lfe1:
	.size	 intArray3D,.Lfe1-intArray3D
	.ident	"GCC: (GNU) 2.95.4  (Debian prerelease)"
```

```
        .file   "array-stack-int3D.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl intArray3D
        .type    intArray3D,@function
intArray3D:
        subl $1228,%esp
        pushl %ebp
        pushl %edi
        pushl %esi
        pushl %ebx
        xorl %ebp,%ebp
        leal 32(%esp),%eax
        movl %eax,24(%esp)
        .p2align 4,,7
.L21:
        xorl %esi,%esi
        leal 1(%ebp),%eax
        movl %eax,28(%esp)
        leal (%ebp,%ebp,4),%eax
        leal (%eax,%eax,4),%ebx
        sall $4,%ebx
        .p2align 4,,7
.L25:
        xorl %ecx,%ecx
        leal 0(,%esi,4),%edx
        leal 1(%esi),%edi
        leal (%esi,%edx),%eax
        sall $3,%eax
        addl %eax,%edx
        addl %ebx,%edx
        addl %ebx,%eax
        .p2align 4,,7
.L29:
        cmpl %ecx,%esi
        jne .L30
        movl %ebp,32(%esp,%edx)
        jmp .L28
        .p2align 4,,7
.L30:
        movl $0,32(%esp,%eax)
.L28:
        addl $4,%eax
        incl %ecx
        cmpl $9,%ecx
        jle .L29
        movl %edi,%esi
        cmpl $9,%esi
        jle .L25
        movl 28(%esp),%ebp
        cmpl $2,%ebp
        jle .L21
        addl $-12,%esp
        movl 36(%esp),%eax
        pushl %eax
        call printArray
        addl $16,%esp
        popl %ebx
        popl %esi
        popl %edi
        popl %ebp
        addl $1228,%esp
        ret
.Lfe1:
        .size    intArray3D,.Lfe1-intArray3D
        .ident  "GCC: (GNU) 2.95.4  (Debian prerelease)"
```

```
#!/usr/bin/perl -w

# Disasm.pl v0.4

#Assumes that the file we're working with is stripped
#TODO:
# 0. Sort NUMERICALLY on function call names, not lexographically
# 1. Get this to work if symbols are present
# 2. Add options or speed up the finding of unused functions (can we do it
#    without an extra pass?)
# 3. Fix various FIXME's
# 4. Make work with sparc

use strict;
use Getopt::Long;

my ($call_graph, $fnames);

if($#ARGV < 0)
{
    print "Usage: $0 <file> [options]\n";
        print "\t--fnames\tprint function names\n";
        print "\t--graph\tgenerate file with graph information for dot\n";
    exit 1;
} else {

        $call_graph = $fnames = 0;
        GetOptions( "fnames"    => \$fnames,     # --fnames
                                "graph"    => \$call_graph   # --graph
                                );

}



my %symbols;

my $fprefix = "function #";
my $lprefix = "label #";

my $return   = "ret    ";
my $call     = "call   0x";
my $jump     = "j..    0x";
my $retsize = 1; #size of ret opcode

#Sparc:
#FIXME: There's a few issues with sparc opcodes:
# 1. We need to handle command line options to specify to use sparc
# 2. We need to allow arbitrary spacing after the branch instruction
# 3. Some functions return with just ret, some return with ret then restore
my $sreturn  = "restore";
my $scall    = "call  ";
my $sjump    = "  b[^t0-9].[^0-9a-f].[ ]*";
my $sparc    = `uname -a` =~ /sparc/;

if($sparc)
{
   $return = $sreturn;
   $call = $scall;
   $jump = $sjump;
}

my $filename = shift(@ARGV);
my @lines = `objdump -TC $filename`;

my %functions;
my %labels;
my %calls;



foreach (@lines)
{
    if(/0([0-9a-f]+).*\*UND\*.* ([^ ]+)$/)
    {
        my $temp = $2;
        chop $temp;
        $symbols{$1} = $temp;
    }
}

@lines = `objdump -dj .text $filename`;


#counters for functions, unused functions, and labels
my $fcount = 1;
my $lcount = 1;

foreach(@lines)
{
    #FIXME: Hack that also assumes stripped binary.. How can we factor this
    #out of the loop?
    if(/([0]+)([0-9a-f]+)\ <.text>/)
    {
   #     print "Text @ $1 $2 ($_)";
        $symbols{$2} = "_start";
        $functions{$2} = "_start";
    }

    if(/$call([0-9a-f]+)/ &&
        ! exists($symbols{$1}))
   # if(/([0-9a-f]+).*$function/)
    {
        $symbols{$1} = "$fprefix$fcount";
        $fcount++;

        $functions{$1} = $symbols{$1};
    }
    elsif(/$jump([0-9a-f]+)/ &&
          ! exists($symbols{$1}))
    {
        $symbols{$1} = "$lprefix$lcount";
        $lcount++;

        $labels{$1} = $symbols{$1};
    }
}

#FIXME: This should be an option...
#
# Nasko - should it? misses some data in the output just uncomment the if
# statement and the corresponding closing brace to make --fnames work
#

my $inFunc;
my $lastRet;
my $storeRet = 0;
my $counter = 0;


# if ($fnames == 1) {

        foreach(@lines)
        {
                ++$counter;
            #HACK: Yeah, this sucks.. but we can't just add 1 to get the next address
            #and I don't know how to peek at the next line
                #
                # Nasko - just use $lines[$counter]
                #
            if($storeRet == 1)
            {
                if(/([0-9a-f]+)/)
                {
                    $lastRet = $1;
                    $storeRet = 0;
                }
                else
                {
                    next;
                }
            }
            if(/([0-9a-f]+)/ and exists($functions{$1}))
            {
                $inFunc = 1;
            }
            elsif(/([0-9a-f]+).*$return/)
            {

                    if($inFunc == 0)
                {
                   $symbols{$lastRet} = "function #$fcount (unused)";
                   $functions{$lastRet} = "function #$fcount (unused)";
                   $fcount++;
                }
                #FIXME: Sure would be nice to peek at the next line and set lastRet
                #right here..
                $storeRet = 1;
                $inFunc = 0;
           }
        }
#}

my $localFunc;
foreach(@lines)
{

    if(/([0-9a-f]+)/ and exists($symbols{$1}))
    {
        my $symb = $symbols{$1};
        if ($symb =~ /label/) {
            $_ = "$symb:\n$_";
        } else {
            $_ = "\n$symb:\n$_";
            $localFunc = $symb;
        }
    }
    elsif(/.*$return/)
    {
        $_ .= "return\n\n";
    }
    elsif(/.*$jump([0-9a-f]+)/ ||
          /.*$call([0-9a-f]+)/)
    {
        chop;
        if(exists($symbols{$1}))
        {
            $_ .= "\t<" . $symbols{$1} . ">\n";
        }
        else
        {
            $_ .= "\t<unknown symbol>\n";
        }

        my $symb = $symbols{$1};

# Why skip labels??
#       if ($symb =~ /label/) {
#           next;
#       }

        if(exists($calls{ $localFunc }))
        {
            push @{$calls{ $localFunc } }, $symb;
        }
        else
        {
            $calls{ $localFunc } = [ $symb ];
        }
    }
    print;
}

print "\nKnown symbols:\n";

foreach (sort (keys %symbols))
{
    if (!($symbols{$_} =~ /label/))
    {
        print;
        print ": $symbols{$_}\n";
    }
}

print "\nCall graph:\n";

my $fName;
if ($call_graph == 1) {
        # a local variable for each function name
        # open the file to store the definition of the graph
        open(FILE, ">call_graph") ||
                die "Couldn't open file for writing the call graph\n";
        print FILE "digraph prof {\n";
}

#foreach (sort keys %calls)
foreach (%calls)
{
        $fName = $_;
    print;
    print ":\n";
    foreach (@{ $calls{$_} })
    {
        my $mytmp = $_;
        if(!($_ =~ /label/))
        {
                print " calls $_\n";

                                # print to the graph file
                if($call_graph == 1) {print FILE "\t\"$fName\" -> \"$_\"\n"};
        }
    }
    print "\n";
}

if ($call_graph == 1) {
        # put the closing brace and close the file
        print FILE "}\n";
        close(FILE);
}
```