Know Your Enemy

Security Warrior

O'REILLY®

Cyrus Peikari & Anton Chuvakin

CHAPTER 3

Linux Reverse Engineering

This chapter is concerned with reverse engineering in the Linux environment, a topic that is still sparsely covered despite years of attention from security consultants, software crackers, programmers writing device drivers or Windows interoperability software. The question naturally arises: why would anyone be interested in reverse engineering on Linux, an operating system in which the applications that are not open source are usually available for no charge? The reason is worth noting: in the case of Linux, reverse engineering is geared toward "real" reverse engineering—such as understanding hardware `ioctl()` interfaces, proprietary network protocols, or potentially hostile foreign binaries—rather than toward the theft of algorithms or bypassing copy protections.

As mentioned in the previous chapter, the legality of software reverse engineering is an issue. While actually illegal in some countries, reverse engineering is for the most part a violation of a software license or contract; that is, it becomes criminal only when the reverse engineer is violating copyright by copying or redistributing copy-protected software. In the United States, the (hopefully temporary) DMCA makes it illegal to circumvent a copy protection mechanism; this means the actual reverse engineering process is legal, as long as protection mechanisms are not disabled. Of course, as shown in the grossly mishandled Sklyarov incident, the feds will go to absurd lengths to prosecute alleged DMCA violations, thereby driving home the lesson that if one is engaged in reverse engineering a copy-protected piece of software, one should not publish the matter. Oddly enough, all of the DMCA cases brought to court have been at the urging of commercial companies...reverse engineering Trojaned binaries, exploits, and viruses seems to be safe for the moment.

This material is not intended to be a magic "Reverse Engineering How-To." In order to properly analyze a binary, you need a broad background in computers, covering not only assembly language but high-level language design and programming, operating system design, CPU architecture, network protocols, compiler design, executable file formats, code optimization—in short, it takes a great deal of experience to know what you're looking at in the disassembly of some random compiled binary.

Little of that experience can be provided here; instead, the standard Linux tools and their usage are discussed, as well their shortcomings. The final half of the chapter is mostly source code demonstrating how to write new tools for Linux.

The information in this chapter may be helpful to software engineers, kernel-mode programmers, security types, and of course reverse engineers and software crackers, who know most of this stuff already. The focus is on building upon or replacing existing tools; everything covered will be available on a standard Linux system containing the usual development tools (gcc, gdb, perl, binutils), although the ptrace section does reference the kernel source at some points.

The reader should have some reasonable experience with programming (shell, Perl, C, and Intel x86 assembler are recommended), a more than passing familiarity with Linux, and an awareness at the very least of what a hex editor is and what it is for.

Basic Tools and Techniques

One of the wonderful things about Unix in general and Linux in particular is that the operating system ships with a number of powerful utilities that can be used for programming or reverse engineering (of course, some commercial Unixes still try to enforce "licensing" of so-called developer tools—an odd choice of phrase since "developers" tend to use Windows and "coders" tend to use Unix—but packages such as the GNU development tools are available for free on virtually every Unix platform extant). A virtual cornucopia of additional tools can be found online (see the "References" section at the end of the chapter), many of which are under continual development.

The tools presented here are restricted to the GNU packages and utilities available in most Linux distributions: nm, gdb, lsof, ltrace, objdump, od, and hexdump. Other tools that have become fairly widely used in the security and reverse engineering fields—dasm, elfdump, hte, ald, IDA, and IDA_Pro—are not discussed, though the reader is encouraged to experiment with them.

One tool whose omission would at first appear to be a matter of great neglect is the humble hex editor. There are many of these available for Linux/Unix. biew is the best; hexedit is supplied with just about every major Linux distribution. Of course, as all true Unixers know in their hearts, you need no hex editor when you're in bed with od and dd.

Overview of the Target

The first tool that should be run on a prospective target is nm, the system utility for listing symbols in a binary. There are quite a few options to nm; the more useful are `-C` (demangle), `-D` (dynamic symbols), `-g` (global/external symbols), `-u` (only undefined symbols), `--defined-only` (only defined symbols), and `-a` (all symbols, including debugger hints).

There are notions of symbol type, scope, and definition in the nm listing. *Type* specifies the section where the symbol is located and usually has one of the following values:

B Uninitialized data (*.bss*)

D Initialized data (*.data*)

N Debug symbol

R Read-only data (*.rodata*)

T Text section/code (*.text*)

U Undefined symbol

W Weak symbol

? Unknown symbol

The *scope* of a symbol is determined by the case of the type; lowercase types are local in scope, while uppercase types are global. Thus, "t" denotes a local symbol in the code section, while "T" denotes a global symbol in the code section. Whether a symbol is *defined* is determined by the type, as listed above; `` `nm -u` `` is equivalent to doing an `` `nm | grep ' \{9,\}[uUwW]'` ``, where the `' \{9,\}'` refers to the empty spaces printed in lieu of an address or value. Thus, in the following example:

```
bash# nm a.out
08049fcc ? _DYNAMIC
08049f88 ? _GLOBAL_OFFSET_TABLE_
08048ce4 R _IO_stdin_used
0804a06c A __bss_start
08049f60 D __data_start
         w __deregister_frame_info@@GLIBC_2.0
08048c90 t __do_global_ctors_aux
         w __gmon_start__
         U __libc_start_main@@GLIBC_2.0
08048cbc ? _fini
08048ce0 R _fp_hw
0804848c ? _init
080485a0 T _start
08048bb4 T bind
080485c4 t call_gmon_start
```

the symbols `_start` and `bind` are exported symbols defined in *.text*; `__do_global_ctors_aux` and `call_gmon_start` are private symbols defined in *.text*, `_DYNAMIC`, `_GLOBAL_OFFSET_TABLE_`, `_fini`, and `_init` are unknown symbols; and `__libc_start_main` is imported from *libc.so*.

Using the proper command switches and filtering based on type, we can see at a glance the layout of the target:

```
List labels in the code sections:
        nm -C --defined-only filename | grep '[0-9a-f ]\{8,\} [Tt]'
```

```
List data:
        nm -C --defined-only filename | grep '[0-9a-f ]\{8,\} [RrBbDd]'
List unresolved symbols [imported functions/variables]:
        nm -Cu
```

The objdump utility also provides a quick summary of the target with its `-f` option:

```
bash# objdump -f /bin/login
/bin/login:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0804a0c0
bash#
```

This is somewhat akin to the `file(1)` command, which has similar output:

```
bash# file /bin/login
/bin/login: setuid ELF 32-bit LSB executable, Intel 80386, version 1,
dynamically linked (uses shared libs), stripped
bash#
```

Both correctly identify the target, though the objdump version gives the BFD target type (see the section "The GNU BFD Library" later in this chapter) as well as the entry point.

The final utility used in the casual assessment of a target is the venerable strings(1), without which the software security industry would apparently curl up and die. The purpose of strings is to print out all ASCII character sequences that are four characters or more long. strings(1) itself is easy to use:

```
List all ASCII strings in the initialized and loaded sections:
        strings -tx
List all ASCII strings in all sections:
        strings -atx
List all ASCII strings that are at least 8 characters in length:
        strings -atx -8
```

It should be noted that the addresses in the "tx" section should be cross-referenced with the address ranges of the various program sections; it is terribly easy to give a false impression about what a program does simply by including data strings such as "setsockopt" and "execve", which can be mistaken for shared library references.

Debugging

Anyone who has spent any reasonable amount of time on a Linux system will be familiar with gdb. The GNU Debugger actually consists of two core components: the console-mode gdb utility, and libgdb, a library intended for embedding gdb in a larger application (e.g., an IDE). Numerous frontends to gdb are available, including ddd, kdbg, gvd, and insight for X-Windows, and vidbg and motor for the console.

As a console-mode program, gdb requires some familiarity on the part of the user; GNU has made available a very useful quick reference card in addition to the copious

"Debugging with GDB" tome (see the "References" section at the end of this chapter for more information).

The first question with any debugger is always "How do you use this to disassemble?" The second follows closely on its heels: "How do you examine memory?" In gdb, we use the `disassemble`, `p` (print), and `x` (examine) commands:

```
disassemble start end :    disasm from 'start' address to 'end'
p $reg                : print contents of register 'reg' ['p $eax']
p address             : print value of 'address' ['p _start']
p *address            : print contents of 'address' ['p *0x80484a0']
x $reg                : disassemble address in 'reg' ['x $eip']
x address             : disassemble 'address' ['x _start']
x *address            : dereference and disassemble address
```

The argument to the `p` and `x` commands is actually an expression, which can be a symbol, a register name (with a "$" prefix), an address, a dereferenced address (with a "*" prefix), or a simple arithmetic expression, such as "$edi + $ds" or "$ebx + ($ecx * 4)".

Both the `p` and `x` commands allow formatting arguments to be appended:

```
x/i    print the result as an assembly language instruction
x/x    print the result in hexadecimal
x/d    print the result in decimal
x/u    print the result in unsigned decimal
x/t    print the result in binary
x/o    print the result in octal
x/f    print the result as a float
x/a    print the result as an address
x/c    print the result as an unsigned char
x/s    print the result as an ASCII string
```

However, `i` and `s` are not usable with the `p` command, as it does not dereference the address it is given.

For examining process data other than address space, gdb provides the `info` command. There are over 30 `info` options, which are documented with the `help info` command; the more useful options are:

```
all-registers      Contents of all CPU registers
args               Arguments for current stack frame [req. syms]
breakpoints        Breakpoint/watch list and status
frame              Summary of current stack frame
functions          Names/addresses of all known functions
locals             Local vars in current stack frame [req. syms]
program            Execution status of the program
registers          Contents of standard CPU registers
set                Debugger settings
sharedlibrary      Status of loaded shared libraries
signals            Debugger handling of process signals
stack              Backtrace of the stack
threads            Threads IDs
tracepoints        Tracepoint list and status
```

```
types               Types recognized by gdb
udot                Kernel user struct for the process
variables           All known global and static variable names
```

Thus, to view the registers, type `info registers`. Many of the `info` options take arguments; for example, to examine a specific register, type `info registers eax`, where `eax` is the name of the register to be examined. Note that the "$" prefix is not needed with the `info register` command.

Now that the state of the process can be easily examined, a summary of the standard process control instructions is in order:

```
continue        Continue execution of target
finish          Execute through end of subroutine (current stack frame)
kill            Send target a SIGKILL
next            Step (over calls) one source line
nexti           Step (over calls) one machine instruction
run             Execute target [uses PTRACE_TRACEME]
step            Step one source line
stepi           Step one machine instruction
backtrace       Print backtrace of stack frames
up              Set scope "up" one stack frame (out of call)
down            Set scope "down" one stack frame (into call)
```

Many of these commands have aliases since they are used so often: `n` (`next`), `ni` (`nexti`), `s` (`step`), `si` (`stepi`), `r` (`run`), `c` (`continue`), and `bt` (`backtrace`).

The use of these commands should be familiar to anyone experienced with debuggers. `stepi` and `nexti` are sometimes referred to as "step into" and "step over," while `finish` is often called "ret" or "p ret." The `backtrace` command requires special attention: it shows how execution reached the current point in the program by analyzing stack frames; the `up` and `down` commands allow the current context to be moved up or down one frame (as far as gdb is concerned, that is; the running target is not affected). To illustrate:

```
gdb> bt
#0  0x804849a in main ( )
#1  0x8048405 in _start ( )
gdb> up
#1  0x8048405 in _start ( )
gdb> down
#0  0x804849a in main ( )
```

The numbers at the start of each line in the backtrace are frame numbers; `up` increments the context frame number (the current frame number is always 0), and `down` decrements it. Details for each frame can be viewed with the `info frame` command:

```
gdb> bt
#0  0x804849a in main ( )
#1  0x8048405 in _start ( )
gdb> info frame 0
Stack frame at 0xbfbffa60:
 eip = 0x804849a in main; saved eip 0x8048405
```

```
 called by frame at 0xbfbffaac
Arglist at 0xbfbffa60, args:
Locals at 0xbfbffa60, Previous frame's sp is 0x0
Saved registers:
 ebp at 0xbfbffa60, eip at 0xbfbffa64
gdb> info frame 1
Stack frame at 0xbfbffaac:
 eip = 0x8048405 in _start; saved eip 0x1
 caller of frame at 0xbfbffa60
Arglist at 0xbfbffaac, args:
Locals at 0xbfbffaac, Previous frame's sp is 0x0
Saved registers:
 ebx at 0xbfbffa94, ebp at 0xbfbffaac, esi at 0xbfbffa98,
 edi at 0xbfbffa9c, eip at 0xbfbffab0
```

It is important to become used to working with stack frames in gdb, as they are likely to be the only frame of reference available while debugging a stripped binary.

A debugger is nothing without breakpoints. Fortunately, gdb provides a rich breakpoint subsystem with support for data and execution breakpoints, commands to execute on breakpoint hits, and breakpoint conditions.

```
break          Set an execution breakpoint
hbreak         Set an execution breakpoint using a debug register
xbreak         Set a breakpoint at the exit of a procedure
clear          Delete breakpoints by target address/symbol
delete         Delete breakpoints by ID number
disable        Disable breakpoints by ID number
enable         Enable breakpoints by ID number
ignore         Ignore a set number of occurrences of a breakpoint
condition      Apply a condition to a breakpoint
commands       Set commands to be executed when a breakpoint hits
```

Each of the break commands takes as its argument a line number, a function name, or an address if prefixed with "*" (e.g., "break *0x8048494"). Conditional breakpoints are supported via the `condition` command of the form:

```
condition num expression
```

...where *num* is the breakpoint ID and *expression* is any expression that evaluates to TRUE (nonzero) in order for the breakpoint to hit; the `break` command also supports an `if` suffix of the form:

```
break address if expression
```

where *expression* is the same as in the command. Breakpoint conditions can be any expression; however, they're devoid of meaning:

```
break main if $eax > 0
break main if *(unsigned long *)(0x804849a +16) == 23
break main if 2 > 1
```

These conditions are associated with a breakpoint number and are deleted when that breakpoint is deleted; alternatively, the condition for a breakpoint can be changed with the `condition` command, or cleared by using the `condition` command with no expression specified.

Breakpoint commands are another useful breakpoint extension. These are specified with `commands`, which has the following syntax:

```
commands num
    command1
    command2
    ...
end
```

num is the breakpoint ID number, and all lines between `commands` and `end` are commands to be executed when the breakpoint hits. These commands can be used to perform calculations, print values, set new breakpoints, or even continue the target:

```
commands 1
info registers
end

commands 2
b *(unsigned long *)$eax
continue
end

commands 3
x/s $esi
x/s $edi
end

commands 4
set $eax = 1
set $eflags = $eflags & ~0x20
set $eflags = $eflags | 0x01
end
```

The last example demonstrates the use of `commands` to set the eax register to 1, to clear the Zero flag, and to set the Carry flag. Any standard C expression can be used in gdb commands.

The `break`, `hbreak`, and `xbreak` commands all have temporary forms that begin with "t" and cause the breakpoint to be removed after it hits. The `tbreak` command, for example, installs an execution breakpoint at the specified address or symbol, then removes the breakpoint after it hits the first time, so that subsequent executions of the same address will not trigger the breakpoint.

This is perhaps a good point to introduce the gdb `display` command. This command is used with an expression (i.e., an address or register) to display a value whenever gdb stops the process, such as when a breakpoint is encountered or an instruction is traced. Unfortunately the `display` command does not take arbitrary gdb commands, so `display info regs` will not work.

It is still useful to display variables or register contents at each stop; this allows "background" watchpoints (i.e., watchpoints that do not stop the process on modification, but are simply displayed) to be set up, and also allows for a runtime context to be displayed:

```
gdb> display/i $eip
gdb> display/s *$edi
gdb> display/s *$esi
gdb> display/t $eflags
gdb> display $edx
gdb> display $ecx
gdb> display $ebx
gdb> display $eax
gdb> n
0x400c58c1 in nanosleep () from /lib/libc.so.6
9: $eax = 0xfffffffc
8: $ebx = 0x4013c0b8
7: $ecx = 0xbffff948
6: $edx = 0x4013c0b8
5: /t $eflags = 1100000010
4: x/s *$esi  0x10000:  <Address 0x10000 out of bounds>
3: x/s *$edi  0xbffffc6f:       "/home/_m/./a.out"
2: x/i $eip  0x400c58c1 <nanosleep+33>:       pop    %ebx
gdb>
```

As can be seen in the above example, the `display` command can take the same formatting arguments as the `p` and `x` commands. A list of all display expressions in effect can be viewed with `info display`, and expressions can be deleted with `undisplay #`, where # is the number of the display as shown in the display listing.

In gdb, a data breakpoint is called a *watchpoint*; a watched address or variable causes execution of the program to stop when the address is read or written. There are three watch commands in gdb:

```
awatch        Set a read/write watchpoint
watch         Set a write watchpoint
rwatch        Set a read watchpoint
```

Watchpoints appear in the breakpoint listing (`info breakpoints`) and are deleted as if they are breakpoints.

One point about breakpoints and watchpoints in gdb on the x86 platform needs to be made clear: the use of x86 debug registers. By default, gdb attempts to use a hardware register for `awatch` and `rwatch` watchpoints in order to avoid slowing down execution of the program; execution breakpoints are embedded INT3 instructions by default, although the `hbreak` is intended to allow hardware register breakpoints on execution access. This support seems to be disabled in many versions of gdb, however; if an `awatch` or `rwatch` cannot be made because of a lack of debug register support, the error message "Expression cannot be implemented with read/access watchpoint" will appear, while if an `hbreak` cannot be installed, the message "No hardware breakpoint support in the target" is printed. The appearance of one of these messages means either that gdb has no hardware debug register support or that all debug registers are in use. More information on Intel debug registers can be found in the sections "Antidebugging" and "Debugging with ptrace," later in this chapter.

One area of debugging with gdb that gets little attention is the support for SIGSTOP via Ctrl-z. Normally, in a terminal application, the shell catches Ctrl-z and the foreground process is sent a SIGSTOP. When gdb is running, however, Ctrl-z sends a SIGSTOP to the target, and control is returned to gdb. Needless to say, this is extremely useful in programs that enter an endless loop, and it can be used as an underpowered replacement for SoftICE's Ctrl-d when debugging an X program from an xterm.

For example, use gdb to run a program with an endless loop:

```
#include <unistd.h>
int main( int argc, char **argv ) {
          int x = 666;
          while ( 1 )  {
                x++;
                sleep(1);
          }
          return(0);
 }

bash# gdb ./a.out
gdb> r
(no debugging symbols found)...(no debugging symbols found)...
```

At this point the program is locked in a loop; press Ctrl-z to stop the program.

```
Program received signal SIGTSTP, Stopped (user).
0x400c58b1 in nanosleep () from /lib/libc.so.6
Program received signal SIGTSTP, Stopped (user).
0x400c58b1 in nanosleep () from /lib/libc.so.6
```

A simple backtrace shows the current location of the program; a judicious application of `finish` commands will step out of the library calls:

```
gdb> bt
#0  0x400c58b1 in nanosleep () from /lib/libc.so.6
#1  0x400c5848 in sleep () from /lib/libc.so.6
#2  0x8048421 in main ()
#3  0x4003e64f in __libc_start_main () from /lib/libc.so.6
gdb> finish
Program received signal SIGTSTP, Stopped (user).
 0x400c58b1 in nanosleep () from /lib/libc.so.6
 gdb> finish
 0x400c5848 in sleep () from /lib/libc.so.6
 gdb> finish
 0x8048421 in main ()
 gdb> dis main
 Dump of assembler code for function main:
 ...
 0x8048414 <main+20>:    incl   0xfffffffc(%ebp)
```

```
0x8048417 <main+23>:    add    $0xfffffff4,%esp
0x804841a <main+26>:    push   $0x1
0x804841c <main+28>:    call   0x80482f0 <sleep>
0x8048421 <main+33>:    add    $0x10,%esp
0x8048424 <main+36>:    jmp    0x8048410 <main+16>
0x8048426 <main+38>:    xor    %eax,%eax
0x8048428 <main+40>:    jmp    0x8048430 <main+48>
0x804842a <main+42>:    lea    0x0(%esi),%esi
0x8048430 <main+48>:    mov    %ebp,%esp
0x8048432 <main+50>:    pop    %ebp
0x8048433 <main+51>:    ret
End of assembler dump.
```

At this point the location of the counter can be seen in the `inc` instruction: `0xfffffffc(%ebp)` or `[ebp-4]` in signed Intel format. A watchpoint can now be set on the counter and execution of the program can be continued with a break each time the counter is incremented:

```
gdb> p $ebp - 4
0xbffffb08
gdb> p/d *($ebp - 4)
$1 = 668
gdb> watch 0xbffffb08
Watchpoint 2: 0xbffffb08
gdb> c
```

Note that the address of the counter on the stack is used for the watch; while a watch could be applied to the `ebp` expression with `watch *($ebp-4)`, this would break whenever the first local variable of a function was accessed—hardly what we want. In general, it is best to place watchpoints on actual addresses instead of variable names, address expressions, or registers.

Now that gdb has been exhaustively introduced, it has no doubt caused the reader some trepidation: while it is powerful, the sheer number of commands is intimidating and makes it hard to use. To overcome this difficulty, you must edit the gdb config file: *~/.gdbinit* on Unix systems. Aliases can be defined between `define` and `end` commands, and commands to be performed at startup (e.g., the `display` command) can be specified as well. Following a sample *.gdbinit*, which should make life easier when using gdb.

First, aliases for the breakpoint commands are defined to make things a bit more regular:

```
# ______________breakpoint aliases_____________
define bpl
 info breakpoints
end

define bpc
 clear $arg0
end
```

```
define bpe
 enable $arg0
end

define bpd
 disable $arg0
end
```

Note that the *.gdbinit* comment character is "#" and that mandatory arguments for a macro can be specified by the inclusion of "$arg#" variables in the macro.

Next up is the elimination of the tedious `info` command; the following macros provide more terse aliases for runtime information:

```
# ______________process information___________
define stack
 info stack
 info frame
 info args
 info locals
end

define reg
 printf "     eax:%08X ebx:%08X  ecx:%08X",  $eax, $ebx, $ecx
 printf " edx:%08X\teflags:%08X\n",  $edx, $eflags
 printf "     esi:%08X edi:%08X  esp:%08X",  $esi, $edi, $esp
 printf " ebp:%08X\teip:%08X\n", $ebp, $eip
 printf "     cs:%04X  ds:%04X  es:%04X", $cs, $ds, $es
 printf "  fs:%04X  gs:%04X  ss:%04X\n", $fs, $gs, $ss
end

define func
 info functions
end

define var
 info variables
end

define lib
 info sharedlibrary
end

define sig
 info signals
end

define thread
 info threads
end

define u
 info udot
end
```

```
define dis
 disassemble $arg0
end

 # ________________hex/ascii dump an address_____________ _
define hexdump
 printf "%08X : ", $arg0
 printf "%02X %02X %02X %02X  %02X %02X %02X %02X",               \
     *(unsigned char*)($arg0), *(unsigned char*)($arg0 + 1),      \
     *(unsigned char*)($arg0 + 2), *(unsigned char*)($arg0 + 3),   \
     *(unsigned char*)($arg0 + 4), *(unsigned char*)($arg0 + 5),   \
     *(unsigned char*)($arg0 + 6), *(unsigned char*)($arg0 + 7)
 printf " - "
 printf "%02X %02X %02X %02X  %02X %02X %02X %02X ",              \
     *(unsigned char*)($arg0 + 8), *(unsigned char*)($arg0 + 9),   \
     *(unsigned char*)($arg0 + 10), *(unsigned char*)($arg0 + 11), \
     *(unsigned char*)($arg0 + 12), *(unsigned char*)($arg0 + 13), \
     *(unsigned char*)($arg0 + 14), *(unsigned char*)($arg0 + 15)
 printf "%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c\n",                        \
     *(unsigned char*)($arg0), *(unsigned char*)($arg0 + 1),      \
     *(unsigned char*)($arg0 + 2), *(unsigned char*)($arg0 + 3),   \
     *(unsigned char*)($arg0 + 4), *(unsigned char*)($arg0 + 5),   \
     *(unsigned char*)($arg0 + 6), *(unsigned char*)($arg0 + 7),   \
     *(unsigned char*)($arg0 + 8), *(unsigned char*)($arg0 + 9),   \
     *(unsigned char*)($arg0 + 10), *(unsigned char*)($arg0 + 11), \
     *(unsigned char*)($arg0 + 12), *(unsigned char*)($arg0 + 13), \
     *(unsigned char*)($arg0 + 14), *(unsigned char*)($arg0 + 15)
end

# ________________process context_____________ _
define context
printf "_______________________________________ _"
printf "________________________________________ _\n"
reg
printf "[%04X:%08X]------------------------", $ss, $esp
printf "---------------------------------[stack]\n"
hexdump $sp+48
hexdump $sp+32
hexdump $sp+16
hexdump $sp
printf "[%04X:%08X]------------------------", $cs, $eip
printf "---------------------------------[ code]\n"
x /8i $pc
printf "----------------------------------------"
printf "---------------------------------------\n"
end
```

Of these, the context macro is the most interesting. This macro builds on the previous reg and hexdump macros, which display the x86 registers and a standard hexadecimal dump of an address, respectively. The context macro formats these and displays an eight-line disassembly of the current instruction.

With the display of information taken care of, aliases can be assigned to the usual process control commands to take advantage of the display macros:

```
# ________________process control______________
define n
 ni
 context
end

define c
 continue
 context
end

define go
 stepi $arg0
 context
end

define goto
 tbreak $arg0
 continue
 context
end

define pret
 finish
 context
end

define start
 tbreak _start
 r
 context
end

define main
 tbreak main
 r
 context
end
```

The `n` command simply replaces the default `step` command with the "step one machine instruction" command and displays the context when the process stops; `c` performs a continue and displays the context at the next process break. The `go` command steps `$arg0` number of instructions, while the `goto` command attempts to execute until address `$arg0` (note that intervening break- and watchpoints will still stop the program), and the `pret` command returns from the current function. Both `start` and `main` are useful for starting a debugging session: they run the target and break on the first execution of `_start()` (the target entry point) and `main()`, respectively.

And, finally, some useful gdb display options can be set:

```
# _________________gdb options_______________ _
set confirm 0
set verbose off
set prompt gdb>
set output-radix 0x10
set input-radix 0x10
```

For brevity, none of these macros provides help text; it can be added using the `document` command to associate a text explanation with a given command:

```
document main
Run program; break on main; clear breakpoint on main
end
```

The text set by the document command will appear under "help user-defined". Using this *.gdbinit*, gdb is finally prepared for assembly language debugging:

```
bash# gdb a.out
   ...
   (no debugging symbols found)...
   gdb> main
   Breakpoint 1 at 0x8048406 in main( )
   ____________________________________________________________________________ _
     eax:00000001 ebx:4013C0B8  ecx:00000000 edx:08048400       eflags:00000282
     esi:40014C34 edi:BFFFFB74  esp:BFFFFAF4 ebp:BFFFFB0C       eip:08048406
     cs:0023  ds:002B  es:002B  fs:0000  gs:0000  ss:002B
   [002B:BFFFFAF4]-----------------------------------------------------[stack]
    BFFFFB3C : 74 FB FF BF  94 E5 03 40 - 80 9F 31 83  04 08 00 84 ...........
    BFFFFB26 : 00 00 48 FB  FF BF 21 E6 - 03 40 00 00  10 83 04 08 ...........
    BFFFFB0A : FF BF 48 FB  FF BF 4F E6 - 03 40 FF BF  7C FB FF BF ...........
    BFFFFAF4 : 84 95 04 08  18 FB FF BF - E8 0F 90 A7  00 40 28 FB ...........
   [0023:08048406]-----------------------------------------------------[ code]
     0x8048406 <main+6>:     movl   $0x29a,0xfffffffc(%ebp)
     0x804840d <main+13>:    lea    0x0(%esi),%esi
     0x8048410 <main+16>:    jmp    0x8048414 <main+20>
     0x8048412 <main+18>:    jmp    0x8048426 <main+38>
     0x8048414 <main+20>:    incl   0xfffffffc(%ebp)
     0x8048417 <main+23>:    add    $0xfffffff4,%esp
     0x804841a <main+26>:    push   $0x1
     0x804841c <main+28>:    call   0x80482f0 <sleep>
   ----------------------------------------------------------------------------
    gdb>
```

The context screen will print in any macro that calls `context` and can be invoked directly if need be; as with typical binary debuggers, a snapshot of the stack is displayed as well as a disassembly of the current instruction and the CPU registers.

Runtime Monitoring

No discussion of reverse engineering tools would be complete without a mention of lsof and ltrace. While neither of these are standard Unix utilities that are guaranteed to ship with a system, they have become quite common and are included in every major Linux distribution as well as FreeBSD, OpenBSD, and NetBSD.

The lsof utility stands for "list open files"; by default, it will display a list of all open files on the system, their type, size, owning user, and the command name and PID of the process that opened them:

```
bash# lsof
COMMAND     PID   USER   FD   TYPE   SIZE      NODE NAME
init          1   root  cwd    DIR    4096         2 /
init          1   root  rtd    DIR    4096         2 /
init          1   root  txt    REG   27856    143002 /sbin/init
init          1   root  mem    REG   92666    219723 /lib/ld-2.2.4.so
init          1   root  mem    REG 1163240    224546 /lib/libc-2.2.4.so
init          1   root   10u  FIFO            64099 /dev/initctl
keventd       2   root  cwd    DIR    4096         2 /
keventd       2   root  rtd    DIR    4096         2 /
keventd       2   root   10u  FIFO             64099 /dev/initctl
ksoftirqd     3   root  cwd    DIR    4096         2 /
...
```

Remember that in Unix, everything is a file; therefore, lsof will list ttys, directories, pipes, sockets, and memory mappings as well as simple files.

The FD or File Descriptor field serves as an identifier and can be used to filter results from the lsof output. FD consists of a file descriptor (a number) or a name, followed by an optional mode character and an optional lock character:

```
10uW       cwd
 ^^---------^^^------------- FD or name
   ^-----------^------------ mode
         ^-----------^----------- lock
```

where *name* is one of:

```
cwd  current working directory
rtd  root dir
pd   parent directory
txt  program [text]
Lnn  library reference
ltx  shared library code [text]
mem  memory-mapped file
```

mode can be one of these:

```
r          read access
w          write access
u          read and write access
space      unknown [no lock character follows]
-          unknown [lock character follows]
```

And lock can be one of:

```
N          Solaris NFS lock [unknown type]
r          read lock [part of file]
R          read lock [entire file]
```

```
w          write lock [part of file]
W          write lock [entire file]
u          read and write lock [any length]
U          unknown lock type
x          SCO OpenServer Xenix lock [part of the file]
X          SCO OpenServer Xenix lock [entire file]
space      no lock
```

The *name* portion of the FD field can be used in conjunction with the `-d` flag to limit the reporting to specific file descriptors:

```
lsof -d 0-3             # List STDIN, STDOUT, STDERR
lsof -d 3-65536         # List all other file descriptors
lsof -d cwd,pd,rtd      # List all directories
lsof -d mem,txt         # List all binaries, libraries, memory maps
```

Specific flags exist for limiting the output to special file types; `-i` shows only TCP/IP sockets, `-U` shows only Unix sockets, and `-N` shows only NFS files:

```
bash# lsof -i
COMMAND     PID   USER   FD   TYPE DEVICE SIZE NODE NAME
inetd     10281   root    4u  IPv4 540746       TCP *:auth (LISTEN)
xfstt     10320   root    2u  IPv4 542171       TCP *:7101 (LISTEN)
bash# lsof -U
COMMAND     PID USER   FD   TYPE     DEVICE SIZE   NODE NAME
gpm         228 root    1u  Unix 0xcf62c3c0         430 /dev/gpmctl
xinit       514   _m    3u  Unix 0xcef05aa0        2357 socket
XFree86     515   _m    1u  Unix 0xcfe0f3e0        2355 /tmp/.X11-Unix/X0
```

To limit the results even further, lsof output can be limited by specifying a PID (process ID) with the `-p` flag, a username with the `-u` flag, or a command name with the `-c` flag:

```
bash# lsof -p 11283
COMMAND   PID USER   FD   TYPE DEVICE    SIZE   NODE NAME
man     11283  man  cwd    DIR    3,1    4096 234285 /usr/share/man
man     11283  man  rtd    DIR    3,1    4096      2 /
man     11283  man  txt    REG    3,1   82848 125776 /usr/lib/man-db/man
...
man     11283  man    3w   REG    3,1   93628 189721 /tmp/zmanoteNaJ
bash# lsof -c snort
COMMAND   PID USER   FD   TYPE DEVICE     NODE NAME
...
snort   10506 root    0u   CHR    1,3    62828 /dev/null
snort   10506 root    1u   CHR    1,3    62828 /dev/null
snort   10506 root    2u   CHR    1,3    62828 /dev/null
snort   10506 root    3u  sock    0,0   546789 can't identify protocol
snort   10506 root    4w   REG    3,1    49916 /var/log/snort/snort.log
```

This can be used effectively with the `-r` command to repeat the listing every *n* seconds; the following example demonstrates updating the listing each second:

```
bash# lsof -c snort -r 1 | grep -v 'REG\|DIR\|CHR'
COMMAND   PID USER   FD   TYPE DEVICE      NODE NAME
snort   10506 root    3u  sock    0,0    546789 can't identify protocol
```

```
=======
COMMAND   PID USER   FD   TYPE DEVICE      NODE NAME
snort   10506 root    3u  sock    0,0    546789 can't identify protocol
=======
...
```

Finally, passing filenames to lsof limits the results to files of that name only:

```
bash# lsof /tmp/zmanoteNaJ
COMMAND   PID USER   FD   TYPE DEVICE  SIZE   NODE NAME
man     11283  man    3w   REG    3,1 93628 189721 /tmp/zmanoteNaJ
sh      11286  man    3w   REG    3,1 93628 189721 /tmp/zmanoteNaJ
gzip    11287  man    3w   REG    3,1 93628 189721 /tmp/zmanoteNaJ
pager   11288  man    3w   REG    3,1 93628 189721 /tmp/zmanoteNaJ
```

Combining this with -r and -o would be extremely useful for tracking reads and writes to a file—if -o was working in lsof.

The ltrace utility traces library and system calls made by a process; it is based on ptrace(), meaning that it can take a target as an argument or attach to a process using the -p PID flag. The flags to ltrace are simple:

```
-p #      Attach to process # and trace
-i        Show instruction pointer at time of call
-S        Show system calls
-L        Hide library calls
-e  list  Include/exclude library calls in 'list'
```

Thus, -L -S shows only the system calls made by the process. The -e parameter takes a comma-separated list of functions to list; if the list is preceded by a "!", the functions are excluded from the output. The list !printf,fprintf prints all library calls except printf() and fprintf(), while -e execl,execlp,execle,execv,execvp prints only the exec calls in the program. System calls ignore the -e lists.

For a library call, ltrace prints the name of the call, the parameters passed to it, and the return value:

```
bash# ltrace -i /bin/date
[08048d01] __libc_start_main(0x080491ec, 1, 0xbffffb44, 0x08048a00,
    0x0804bb7c <unfinished ...>
[08048d89] __register_frame_info(0x0804ee94, 0x0804f020, 0xbffffae8,
    0x40050fe8, 0x4013c0b8) = 0x4013cde0
...
[0804968e] time(0xbffffa78)                          = 1039068300
[08049830] localtime(0xbffffa38)                     = 0x401407e0
[0804bacd] realloc(NULL, 200)                        = 0x0804f260
[080498b8] strftime("Wed Dec  4 22:05:00 PST 2002", 200,
    "%a %b %e %H:%M:%S %Z %Y", 0x401407e0) = 28
[080498d2] printf("%s\n", "Wed Dec  4 22:05:00 PST 2002") = 29
```

System call traces have similar parameters, although the call names are preceded by "SYS_", and the syscall ordinal may be present if the name is unknown:

```
bash# ltrace -S -L /bin/date
SYS_uname(0xbffff71c)                                = 0
```

```
SYS_brk(NULL)                                        = 0x0804f1cc
SYS_mmap(0xbffff50c, 0x40014ea0, 0x400146d8, 4096, 640) = 0x40015000
...
SYS_time(0xbffffa78, 0x0804ca74, 0, 0, 0)            = 0x3deeeba0
SYS_open("/etc/localtime", 0, 0666)                  = 3
SYS_197(3, 0xbffff75c, 0x4013ce00, 0x4014082c, 3) = 0
SYS_mmap(0xbffff724, 0xbffff75c, 0x4013c0b8, 0x0804f220, 4096)=0x40016000
SYS_read(3, "TZif", 4096)                            = 1017
SYS_close(3)                                         = 0
SYS_munmap(0x40016000, 4096)                         = 0
SYS_197(1, 0xbffff2ac, 0x4013ce00, 0x4014082c, 1) = 0
SYS_ioctl(1, 21505, 0xbffff1f8, 0xbffff240, 8192) = 0
SYS_mmap(0xbffff274, 0, 0x4013c0b8, 0x401394c0, 4096) = 0x40016000
SYS_write(1, "Wed Dec  4 22:01:04 PST 2002\n", 29) = 29
...
```

The ltrace utility is extremely useful when attempting to understand a target; however, it must be used with caution, for it is trivial for a target to detect if it is being run under ptrace. It is advisable to always run a potentially hostile target under a debugger such as gdb before running it under an automatic trace utility such as ltrace; this way, any ptrace-based protections can be observed and countered in preparation for the ltrace.

Disassembly

The disassembler is the most important tool in the reverse engineer's kit; without it, automatic analysis of the target is difficult, if not impossible. The good news is that Unix and Linux systems ship with a working disassembler; unfortunately, it is not a very good one. The objdump utility is usually described as "sufficient"; it is an adequate disassembler, with support for all of the file types and CPU architectures that the BFD library understands (see the section "The GNU BFD Library"). Its analysis is a straightforward sequential disassembly; no attempt is made to reconstruct the control flow of the target. In addition, it cannot handle binaries that have missing or invalid section headers, such as those produced by sstrip (see the upcoming "Antidisassembly" section).

It should be made clear that a disassembler is a utility that converts the machine-executable binary code of a program into the human-readable assembly language for that processor. In order to make use of a disassembler, you must have some familiarity with the assembly language to which the target will be converted. Those unfamiliar with assembly language and how Linux programs written in assembly language look are directed to read the available tutorials and source code (see the "References" section).

The basic modes of objdump determine its output:

```
objdump -f [target]    Print out a summary of the target
objdump -h [target]    Print out the ELF section headers
objdump -p [target]    Print out the ELF program headers
objdump -T [target]    Print out the dynamic symbols [imports]
```

```
objdump -t [target]    Print out the local symbols
objdump -d [target]    Disassemble all code sections
objdump -D [target]    Disassemble all sections
objdump -s [target]    Print the full contents of all sections
```

Details of the ELF headers are discussed further under "The ELF File Format."

When in one of these modes, objdump can print out specific ELF sections with the `-j` argument:

```
objdump -j [section-name] [target]
```

Note that *section-name* can only refer to sections in the section headers; the segments in the program headers cannot be dumped with the `-j` flag. The `-j` flag is useful for limiting the output of objdump to only the desired sections (e.g., in order to skip the dozens of compiler version strings that GCC packs into each object file). Multiple `-j` flags have no effect; only the last `-j` flag is used.

The typical view of a target is that of a file header detailing the sections in the target, followed by a disassembly of the code sections and a hex dump of the data sections. This can be done easily with multiple `objdump` commands:

```
bash# (objdump -h a.out; objdump -d a.out; objdump -s i-j .data;       \
      objdump -s -j .rodata) > a.out.lst
```

By default, objdump does not show hexadecimal bytes, and it skips blocks of NULL bytes when disassembling. This default behavior may be overridden with the `--show-raw-insn` and `--disassemble-zeroes` options.

Hex Dumps

In addition to the objdump disassembler, Unix and Linux systems ship with the octal dump program, or od. This is useful when a hex, octal, or ASCII dump of a program is needed; for example, when objdump is unable to process the file or when the user has scripts that will process binary data structures found in the data sections. The data addresses to be dumped can be obtained from objdump itself by listing the program headers and using grep to filter the listing:

```
bash# objdump -h a.out | grep "\.rodata\|\.data" |          \
      awk '{ printf("-j 0x%s -N 0x%s a.out\n", $6, $3)  }' | \
    xargs -n 5 -t od -A x -t x1 -t c -w16

od -A x -t x1 -t c -w16 a.out -j 0x00001860 -N 0x00000227
001860 03 00 00 00 01 00 02 00 00 00 00 00 00 00 00 00
       003  \0  \0  \0 001  \0 002  \0  \0  \0  \0  \0  \0  \0  \0  \0
001870 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
001880 44 65 63 65 6d 62 65 72 00 4e 6f 76 65 6d 62 65
         D   e   c   e   m   b   e   r  \0   N   o   v   e   m   b   e
...
od -A x -t x1 -t c -w16 a.out -j 0x00001aa0 -N 0x00000444
001aa0 00 00 00 00 f4 ae 04 08 00 00 00 00 00 00 00 00
        \0  \0  \0  \0 364 256 004  \b  \0  \0  \0  \0  \0  \0  \0  \0
```

```
001ab0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
001ac0 40 28 23 29 20 43 6f 70 79 72 69 67 68 74 20 28
        @   (   #   )       C   o   p   y   r   i   g   h   t       (
...
```

The `xargs -t` option prints the full `od` command before displaying the output; the arguments passed to `od` in the above example are:

```
-A x        Use hexadecimal ['x'] for the address radix in output
-t x1       Print the bytes in one-byte ['1'] hex ['x'] format
-t c        Print the character representation of each byte
-w16        Print 16 bytes per line
-j addr     Start at offset 'addr' in the file
-N len      Print up to 'len' bytes from the start of the file
```

The output from the above example could be cleaned up by removing the `-t c` argument from `od` and the `-t` argument from `xargs`.

In some systems, od has been replaced by hexdump, which offers much more control over formatting—at the price of being somewhat complicated.

```
bash# objdump -h a.out | grep "\.rodata\|\.data" |              \
    awk '{ off = sprintf( "0x%s", $6 ); len = sprintf( "0x%s", $3);  \
    printf("-s %s -n %d a.out\n", off, len)  }' |                   \
    xargs -n 5 -t hexdump -e                                       \
    '"%08_ax: " 8/1 "%02x " " - " 8/1  "%02x " "  "'                \
    -e '"%_p"'  '"\n"'
```

The hexdump arguments appear more complex than those to od due to the format string passed; however, they are very similar:

```
-s addr    Start at offset 'addr' in the file
-n len     Print up to 'len' bytes from the start of the file
-e format
```

The hexdump format string is `fprintf()` inspired, but it requires some maniacal quoting to make it functional. The formatting codes take the format *`iteration_count/byte_count`* `"format_str"`, where “iteration_count” is the number of times to repeat the effect of the format string, and “byte_count” is the number of data bytes to use as input to the format string. The format strings used in the above example are:

```
%08_ax Print address of byte with field width of 8
%02x   Print hex value of byte with field width of 2
%p     Print ASCII character of next byte or '.'
```

These are strung together with string constants such as ” “, ” - “, and “\n”, which will be printed between the expansion of the formatting codes. The example uses three format strings to ensure that the ASCII representation does not throw off the byte count; thus, the first format string contained within protective single-quotes consists of an address, eight 1-byte %02x conversions, a space/hyphen delimiter, eight more 1-byte %02x conversions, and a space delimiter; the second consists of an ASCII conversion on the same set of input, and the third ignores the set of input and printf a newline. All format strings are applied in order.

Note that unlike od, hexdump does not take hex values as input for its `len` parameter; a bit of awk manipulation was performed on the input to acquire correct input values. The output from hexdump is worth the extra complexity:

```
bash# hexdump -e '"%08_ax: " 8/1 "%02x " " - " 8/1  "%02x " "  "' -e '"%_p"' \
  -e '"\n"' -s 0x00001860 -n 551 a.out
00001860: 03 00 00 00 01 00 02 00 - 00 00 00 00 00 00 00 00  ................
00001870: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
00001880: 44 65 63 65 6d 62 65 72 - 00 4e 6f 76 65 6d 62 65  December.Novembe
...
bash# hexdump -e '"%08_ax: " 8/1 "%02x " " - " 8/1  "%02x " "  "' -e '"%_p"' \
  -e '"\n"' -s 0x00001aa0 -n 1092 a.out
00001aa0: 00 00 00 00 f4 ae 04 08 - 00 00 00 00 00 00 00 00  ................
00001ab0: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
00001ac0: 40 28 23 29 20 43 6f 70 - 79 72 69 67 68 74 20 28  @(#) Copyright (
...
```

The output of either od or hexdump can be appended to an objdump disassembly in order to provide a more palatable data representation than `objdump -s`, or can be passed to other Unix utilities in order to scan for strings or patterns of bytes or to parse data structures.

A Good Disassembly

The output of objdump leaves a little to be desired. In addition to being a "dumb" or *sequential* disassembler, it provides very little information that can be used to understand the target. For this reason, a great deal of post-disassembly work must be performed in order to make a disassembly useful.

Identifying Functions

As a disassembler, objdump does not attempt to identify functions in the target; it merely creates code labels for symbols found in the ELF header. While it may at first seem appropriate to generate a function for every address that is called, this process has many shortcomings; for example, it fails to identify functions only called via pointers or to detect a "call 0x0" as a function.

On the Intel platform, functions or subroutines compiled from a high-level language usually have the following form:

```
55         push ebp
89 E5        movl %esp, %ebp
83 EC ??    subl ??, %esp
...
89 EC        movl %ebp, %esp        ; could also be C9 leave
C3        ret
```

The series of instructions at the beginning and end of a function are called the function *prologue* and *epilogue*; they are responsible for creating a stack frame in which the function will execute, and are generated by the compiler in accordance with the calling convention of the programming language. Functions can be identified by searching for function prologues within the disassembled target; in addition, an arbitrary series of bytes could be considered code if it contains instances of the 55 89 E5 83 EC byte series.

Intermediate Code Generation

Performing automatic analysis on a disassembled listing can be quite tedious. It is much more convenient to do what more sophisticated disassemblers do: translate each instruction to an intermediate or internal representation and perform all analyses on that representation, converting back to assembly language (or to a higher-level language) before output.

This intermediate representation is often referred to as *intermediate code*; it can consist of a compiler language such as the GNU RTL, an assembly language for an idealized (usually RISC) machine, or simply a structure that stores additional information about the instruction.

The following Perl script generates an intermediate representation of objdump output and a hex dump; instructions are stored in lines marked “INSN”, section definitions are stored in lines marked “SEC”, and the hexdump is stored in lines marked “DATA”.

```
#---------------------------------------------------------------------------
    #!/usr/bin/perl
    # int_code.pl : Intermediate code generation based on objdump output
    # Output Format:
    # Code:
    # INSN|address|name|size|hex|mnemonic|type|src|stype|dest|dtype|aux|atype
    # Data:
    # DATA|address|hex|ascii
    # Section Definition:
    # SEC|name|size|address|file_offset|permissions

    my $file = shift;
    my $addr, $hex, $mnem, $size;
    my $s_type, $d_type, $a_type;
    my $ascii, $pa, $perm;
    my @ops;

    if (! $file ) {
        $file = "-";
    }
    open( A, $file ) || die "unable to open $file\n";

    foreach (<A>) {
        # is this data?
        if ( /^([0-9a-fA-F]{8,})\s+                   # address
            (([0-9a-fA-f]{2,}\s{1,2}){1,16})\s*       # 1-16 hex bytes
            \|([^|]{1,16})\|                          # ASCII chars in ||
                                    /x) {
```

```
    $addr = $1;
    $hex = $2;
    $ascii = $4;
    $hex =~ s/\s+/ /g;
    $ascii =~ s/\|/./g;
    print "DATA|$addr|$hex|$ascii\n";
# Is this an instruction?
}elsif ( /^\s?(0x0)?([0-9a-f]{3,8}):?\s+        # address
        (([0-9a-f]{2,}\s)+)\s+                  # hex bytes
        ([a-z]{2,6})\s+                         # mnemonic
        ([^\s].+)                               # operands
                                $/x) {
    $addr = $2;
    $hex = $3;
    $mnem = $5;

    @ops = split_ops($6);

    $src = $ops[0];
    $dest = $ops[1];
    $aux = $ops[2];

    $m_type = insn_type( $mnem );
    if ( $src ) {
        $s_type = op_type( \$src );
    }
    if ( $dest ) {
        $d_type = op_type( \$dest );
    }
    if ( $aux ) {
        $a_type = op_type( \$aux );
    }

    chop $hex;    # remove trailing ' '
    $size = count_bytes( $hex );
    print "INSN|";            # print line type
    print "$addr|$name|$size|$hex|";
    print "$mnem|$m_type|";
    print "$src|$s_type|$dest|$d_type|$aux|$a_type\n";
    $name = "";    # undefine name
    $s_type = $d_type = $a_type = "";
# is this a section?
} elsif ( /^\s*[0-9]+\s                # section number
        ([.a-zA-Z_]+)\s+               # name
        ([0-9a-fA-F]{8,})\s+           # size
        ([0-9a-fA-F]{8,})\s+           # VMA
        [0-9a-fA-F]{8,}\s+             # LMA
        ([0-9a-fA-F]{8,})\s+           # File Offset
                          /x) {
    $name = $1;
    $size = $2;
    $addr = $3;
    $pa = $4;
```

```
        if ( /LOAD/ ) {
            $perm = "r";
            if ( /CODE/ ) {
                $perm .= "x";
                } else {
                $perm .= "-";
            }
            if ( /READONLY/ ) {
                $perm .= "-";
            } else {
                $perm .= "w";
            }
        } else {
            $perm = "---";
        }
        print "SEC|$name|$size|$addr|$pa|$perm\n";
    } elsif ( /^[0-9a-f]+\s+<([a-zA-Z._0-9]+)>:/) {
        # is this a name? if so, use for next addr
        $name = $1;
    } # else ignore line
}
close (A);


sub insn_in_array {
    my ($insn, $insn_list) = @_;
    my $pattern;

    foreach( @{$insn_list} ) {
        $pattern = "^$_";
        if ( $insn =~ /$pattern/ ) {
            return(1);
        }
    }
    return(0);
}

sub insn_type {
    local($insn) = @_;
    local($insn_type) = "INSN_UNK";
    my @push_insns = ("push");
    my @pop_insns = ("pop");
    my @add_insns = ("add", "inc");
    my @sub_insns = ("sub", "dec", "sbb");
    my @mul_insns = ("mul", "imul", "shl", "sal");
    my @div_insns = ("div", "idiv", "shr", "sar");
    my @rot_insns = ("ror", "rol");
    my @and_insns = ("and");
    my @xor_insns = ("xor");
    my @or_insns = ("or");
    my @jmp_insns = ("jmp", "ljmp");
    my @jcc_insns = ("ja", "jb", "je", "jn", "jo", "jl", "jg", "js",
                     "jp");
    my @call_insns = ("call");
```

```
    my @ret_insns = ("ret");
    my @trap_insns = ("int");
    my @cmp_insns = ("cmp", "cmpl");
    my @test_insns = ("test", "bt");
    my @mov_insns = ("mov", "lea");

    if (insn_in_array($insn, \@jcc_insns) == 1) {
        $insn_type = "INSN_BRANCHCC";
    } elsif ( insn_in_array($insn, \@push_insns) == 1 ) {
        $insn_type = "INSN_PUSH";
    } elsif ( insn_in_array($insn, \@pop_insns) == 1 ) {
        $insn_type = "INSN_POP";
    } elsif ( insn_in_array($insn, \@add_insns) == 1 ) {
        $insn_type = "INSN_ADD";
    } elsif ( insn_in_array($insn, \@sub_insns) == 1 ) {
        $insn_type = "INSN_SUB";
    } elsif ( insn_in_array($insn, \@mul_insns) == 1 ) {
        $insn_type = "INSN_MUL";
    } elsif ( insn_in_array($insn, \@div_insns) == 1 ) {
        $insn_type = "INSN_DIV";
    } elsif ( insn_in_array($insn, \@rot_insns) == 1 ) {
        $insn_type = "INSN_ROT";
    } elsif ( insn_in_array($insn, \@and_insns) == 1 ) {
        $insn_type = "INSN_AND";
    } elsif ( insn_in_array($insn, \@xor_insns) == 1 ) {
        $insn_type = "INSN_XOR";
    } elsif ( insn_in_array($insn, \@or_insns) == 1 ) {
        $insn_type = "INSN_OR";
    } elsif ( insn_in_array($insn, \@jmp_insns) == 1 ) {
        $insn_type = "INSN_BRANCH";
    } elsif ( insn_in_array($insn, \@call_insns) == 1 ) {
        $insn_type = "INSN_CALL";
    } elsif ( insn_in_array($insn, \@ret_insns) == 1 ) {
        $insn_type = "INSN_RET";
    } elsif ( insn_in_array($insn, \@trap_insns) == 1 ) {
        $insn_type = "INSN_TRAP";
    } elsif ( insn_in_array($insn, \@cmp_insns) == 1 ) {
        $insn_type = "INSN_CMP";
    } elsif ( insn_in_array($insn, \@test_insns) == 1 ) {
        $insn_type = "INSN_TEST";
    } elsif ( insn_in_array($insn, \@mov_insns) == 1 ) {
        $insn_type = "INSN_MOV";
    }
    $insn_type;
}

sub op_type {
    local($op) = @_; # passed as reference to enable mods
    local($op_type) = "";

    # strip dereference operator
    if ($$op =~ /^\*(.+)/ ) {
        $$op = $1;
    }
```

```
        if ( $$op =~ /^(\%[a-z]{2,}:)?(0x[a-f0-9]+)?\([a-z\%,0-9]+\)/ ) {
            # Effective Address, e.g., [ebp-8]
            $op_type = "OP_EADDR";
        } elsif ( $$op =~ /^\%[a-z]{2,3}/ ) {
            # Register, e.g.,, %eax
            $op_type = "OP_REG";
        } elsif ( $$op =~ /^\$[0-9xXa-f]+/ ) {
            # Immediate value, e.g., $0x1F
            $op_type = "OP_IMM";
        } elsif ( $$op =~ /^0x[0-9a-f]+/ ) {
            # Address, e.g., 0x8048000
            $op_type = "OP_ADDR";
        } elsif ( $$op =~ /^([0-9a-f]+)\s+<[^>]+>/ ) {
            $op_type = "OP_ADDR";
            $$op = "0x$1";
        } elsif ( $$op ne "" )  {
            # Unknown operand type
            $op_type = "OP_UNK";
        }
        $op_type;
    }

    sub split_ops {
        local($opstr) = @_;
        local(@op);

        if ( $opstr =~ /^([^\(]*\([^\)]+\)),\s?        # effective addr
                  (([a-z0-9\%\$_]+)(,\s?               # any operand
                  (.+))?)?                             # any operand
                                    /x ) {
            $op[0] = $1;
            $op[1] = $3;
            $op[2] = $5;
        } elsif ( $opstr =~ /^([a-z0-9\%\$_]+),\s?    # any operand
                      ([^\(]*\([^\)]+\))(,\s?         # effective addr
                      (.+))?                          # any operand
                                    /x ) {
            $op[0] = $1;
            $op[1] = $2;
            $op[2] = $4;
        } else {
            @op = split ',', $opstr;
        }
        @op;
    }

    sub count_bytes {
        local(@bytes) = split ' ', $_[0];
        local($len) = $#bytes + 1;
        $len;
    }
#---------------------------------------------------------------------------
```

The instruction types in this script are primitive but adequate; they can be expanded as needed to handle unrecognized instructions.

By combining the output of objdump with the output of a hexdump (here the BSD utility hd is simulated with the `hexdump` command, using the format strings `-e '"%08_ax: " 8/1 "%02x " " - " 8/1 "%02x " " |"' -e '"%_p"' -e '"|\n"'` mentioned in the "Hex Dumps" section), a complete representation of the target can be passed to this script for processing:

```
bash# (objdump -hw -d a.out; hd a.out) | ./int_code.pl
```

This writes the intermediate code to STDOUT; the intermediate code can be written to a file or piped to other utilities for additional processing. Note that lines for sections, instructions, and data are created:

```
SEC|.interp|00000019|080480f4|000000f4|r--
SEC|.hash|00000054|08048128|00000128|r--
SEC|.dynsym|00000100|0804817c|0000017c|r--
...
INSN|80484a0|_fini|1|55|push|INSN_PUSH|%ebp|OP_REG||||
INSN|80484a1||2|89 e5|mov|INSN_MOV|%esp|OP_REG|%ebp|OP_REG||
INSN|80484a3||3|83 ec 14|sub|INSN_SUB|$0x14|OP_IMM|%esp|OP_REG||
INSN|80484a6||1|53|push|INSN_PUSH|%ebx|OP_REG||||
INSN|80484a7||5|e8 00 00 00 00|call|INSN_CALL|0x80484ac|OP_ADDR||||
INSN|80484ac||1|5b|pop|INSN_POP|%ebx|OP_REG||||
INSN|80484ad||6|81 c3 54 10 00 00|add|INSN_ADD|$0x1054|OP_IMM|%ebx|OP_REG||
INSN|80484b4||5|e8 a7 fe ff ff|call|INSN_CALL|0x8048360|OP_ADDR||||
INSN|80484b9||1|5b|pop|INSN_POP|%ebx|OP_REG||||
...
DATA|00000000|7f 45 4c 46 01 01 01 09 00 00 00 00 00 00 00 00 |.ELF...........
DATA|00000010|02 00 03 00 01 00 00 00 88 83 04 08 34 00 00 00 |...........4...
```

The first field of each line gives the type of information stored in a line. This makes it possible to expand the data file in the future with lines such as TARGET, NAME, LIBRARY, XREF, STRING, and so forth. The scripts in this section will only make use of the INSN information; all other lines are ignored.

When the intermediate code has been generated, the instructions can be loaded into a linked list for further processing:

```
#---------------------------------------------------------------------------
    #!/usr/bin/perl
    # insn_list.pl -- demonstration of instruction linked list creation

    my $file = shift;
    my $insn, $prev_insn, $head;
    if (! $file ) {
            $file = "-";
    }
    open( A, $file ) || die "unable to open $file\n";

    foreach (<A>) {
    if ( /^INSN/ ) {
```

```
        chomp;
        $insn = new_insn( $_ );

        if ( $prev_insn ) {
            $$insn{prev} = $prev_insn;
            $$prev_insn{next} = $insn;
        } else {
            $head = $insn;
        }
        $prev_insn = $insn;
    } else {
        print;
    }
}
close (A);


$insn = $head;
while ( $insn ) {
    # insert code to manipulate list here
     print "insn $$insn{addr} : ";
     print "$$insn{mnem}\t$$insn{dest}\t$$insn{src}\n";
     $insn = $$insn{next};
}

# generate new instruction struct from line
sub new_insn {
    local($line) = @_;
    local(%i, $jnk);
    # change this when input file format changes!
    ( $jnk, $i{addr}, $i{name}, $i{size}, $i{bytes},
      $i{mnem}, $i{mtype}, $i{src}, $i{stype},
      $i{dest}, $i{dtype}, $i{arg}, $i{atype} ) =
        split '\|', $line;
    return \%i;
}
#---------------------------------------------------------------------------
```

The intermediate form of disassembled instructions can now be manipulated by adding code to the `while ( $insn )` loop. As an example, the following code creates cross-references:

```
#-------------------------------------------------------------------------------
# insn_xref.pl -- generate xrefs for data from int_code.pl
# NOTE: this changes the file format to
# INSN|addr|name|size|bytes|mem|mtyp|src|styp|dest|dtype|arg|atyp|xrefs

my %xrefs;    # add this global variable

# new version of while (insn) loop
$insn = $head;
while ( $insn ) {
     gen_xrefs( $insn, $$insn{src}, $$insn{stype} );
     gen_xrefs( $insn, $$insn{dest}, $$insn{dtype} );
```

```
        gen_xrefs( $insn, $$insn{arg}, $$insn{atype} );
        $insn = $$insn{next};
    }

    # output loop
    $insn = $head;
    while ( $insn ) {
        if ( $xrefs{$$insn{addr}} ) {
            chop $xrefs{$$insn{addr}};    # remove trailing colon
        }
        print "INSN|";                    # print line type
        print "$$insn{addr}|$$insn{name}|$$insn{size}|$$insn{bytes}|";
        print "$$insn{mnem}|$$insn{mtype}|$$insn{src}|$$insn{stype}|";
        print "$$insn{dest}|$$insn{dtype}|$$insn{arg}|$$insn{atype}|";
        print "$xrefs{$$insn{addr}}\n";
        $insn = $$insn{next};
    }

    sub gen_xrefs {
        local($i, $op, $op_type) = @_;
        local $addr;
        if ( $op_type eq "OP_ADDR" && $op =~ /0[xX]([0-9a-fA-F]+)/ ) {
            $addr = $1;
            $xrefs{$addr} .= "$$i{addr}:";
        }
        return;
    }
    #---------------------------------------------------------------------------
```

Naturally, there is much more that can be done aside from merely tracking cross-references. The executable can be scanned for strings and address references for them created, system and library calls can be replaced with their C names and prototypes, DATA lines can be fixed to use RVAs instead of file offsets using information in the SEC lines, and higher-level language constructs can be generated.

Such features can be implemented with additional scripts that print to STDOUT a translation of the input (by default, STDIN). When all processing is finished, the intermediate code can be printed using a custom script:

```
    #-------------------------------------------------------------------------------
        #!/usr/bin/perl
        # insn_output.pl -- print disassembled listing
        #                   NOTE: this ignores SEC and DATA lines

        my $file = shift;
        my %insn, $i;
        my @xrefs, $xrefstr;
        if (! $file ) {
                $file = "-";
        }
        open( A, $file ) || die "unable to open $file\n";

        foreach (<A>) {
```

```
    if ( /^INSN|/ ) {
        chomp;
        $i = new_insn( $_ );
        $insn{$$i{addr}} = $i;
    } else {
        ; # ignore other lines
    }
}
close (A);


foreach ( sort keys %insn ) {
    $i = $insn{$_};
    $xrefstr = "";
    @xrefs = undef;
    if ($$i{name}) {
        print "\n$$i{name}:\n";
    } elsif ( $$i{xrefs} ) {
        # generate fake name
        print "\nloc_$$i{addr}:\n";
        @xrefs = split ':', $$i{xrefs};
        foreach ( @xrefs ) {
            $xrefstr .= " $_";
        }
    }
    print "\t$$i{mnem}\t";
    if ( $$i{src} ) {
        print_op( $$i{src}, $$i{stype} );
        if ( $$i{dest} ) {
            print ", ";
            print_op( $$i{dest}, $$i{dtype} );
            if ( $$i{arg} ) {
                print ", ";
                print_op( $$i{arg}, $$i{atype} );
            }
        }
    }
    print "\t\t(Addr: $$i{addr})";
    if ( $xrefstr ne "" ) {
        print " References:$xrefstr";
    }
    print "\n";
}

sub print_op {
    local($op, $op_type) = @_;
    local $addr, $i;
    if ( $op_type eq "OP_ADDR" && $op =~ /0[xX]([0-9a-fA-F]+)/ ) {
        # replace addresses with their names
        $addr = $1;
        $i = $insn{$addr};
        if ( $$i{name} ) {
            print "$$i{name}";
        } else {
```

```
                print "loc_$addr";
            }
        } else {
            print "$op";
        }
        return;
    }

    # generate new instruction struct from line
    sub new_insn {
        local($line) = @_;
        local(%i, $jnk);
        # change this when input file format changes!
        ( $jnk, $i{addr}, $i{name}, $i{size}, $i{bytes},
          $i{mnem}, $i{mtype}, $i{src}, $i{stype},
          $i{dest}, $i{dtype}, $i{arg}, $i{atype}, $i{xrefs} ) =
            split '\|', $line;
        return \%i;
    }
#---------------------------------------------------------------------------
```

This can receive the output of the previous scripts from STDIN:

```
bash# (objdump -hw -d a.out, hd a.out) | int_code.pl | insn_xref.pl \
| insn_output.pl
```

In this way, a disassembly tool chain can be built according to the standard Unix model: many small utilities performing simple transforms on a global set of data.

Program Control Flow

One of the greatest advantages of reverse engineering on Linux is that the compiler and libraries used to build the target are almost guaranteed to be the same as the compiler and libraries that are installed on your system. To be sure, there are version differences as well as different optimization options, but generally speaking all programs will be compiled with gcc and linked with glibc. This is an advantage because it makes it possible to guess what higher-level language constructs caused a particular set of instructions to be generated.

The code generated for a series of source code statements can be determined by compiling those statements in between a set of assembly language markers—uncommon instructions that make the compiled code stand out:

```
#define MARKER asm("\tint3\n\tint3\n\tint3\n");

int main( int argc, char **argv ) {
    int x, y;
    MARKER
    /* insert code to be tested here */
    MARKER
    return(0);
};
```

One of the easiest high-level constructs to recognize is the WHILE loop, due to its distinct backward jump. In general, any backward jump that does not exceed the bounds of a function (i.e., a jump to an address in memory before the start of the current function) is indicative of a loop.

The C statement:

```
while ( x < 1024 ) { y += x; }
```

compiles to the following assembly under gcc:

```
80483df:      cc                      int3
80483e0:      81 7d fc ff 03 00 00    cmpl   $0x3ff,0xfffffffc(%ebp)
80483e7:      7e 07                   jle    80483f0 <main+0x20>
80483e9:      eb 0d                   jmp    80483f8 <main+0x28>
80483eb:      90                      nop
80483ec:      8d 74 26 00             lea    0x0(%esi,1),%esi
80483f0:      8b 45 fc                mov    0xfffffffc(%ebp),%eax
80483f3:      01 45 f8                add    %eax,0xfffffff8(%ebp)
80483f6:      eb e8                   jmp    80483e0 <main+0x10>
```

By removing statement-specific operands and instructions, this can be reduced to the more general pattern:

```
; WHILE
L1:
    cmp    ?, ?
    jcc    L2    ; jump to loop body
    jmp    L3    ; exit from loop
L2    :
    ?    ?, ?    ; body of WHILE loop
    jmp    L1    ; jump to start of loop
; ENDWHILE
L3:
```

where `jcc` is one of the Intel conditional branch instructions.

A related construct is the FOR loop, which is essentially a WHILE loop with a counter. Most C FOR loops can be rewritten as WHILE loops by adding an initialization statement, a termination condition, and a counter increment.

The C FOR statement:

```
for ( x > 0; x < 10; x++ ) { y *= 1024; }
```

is compiled by gcc to:

```
80483d9:      8d b4 26 00 00 00 00    lea    0x0(%esi,1),%esi
80483e0:      83 7d fc 09             cmpl   $0x9,0xfffffffc(%ebp)
80483e4:      7e 02                   jle    80483e8 <main+0x18>
80483e6:      eb 18                   jmp    8048400 <main+0x30>
80483e8:      8b 45 f8                mov    0xfffffff8(%ebp),%eax
80483eb:      89 c2                   mov    %eax,%edx
80483ed:      89 d0                   mov    %edx,%eax
80483ef:      c1 e0 0a                shl    $0xa,%eax
80483f2:      89 45 f8                mov    %eax,0xfffffff8(%ebp)
```

```
80483f5:       ff 45 fc                incl   0xfffffffc(%ebp)
80483f8:       eb e6                   jmp    80483e0 <main+0x10>
80483fa:       8d b6 00 00 00 00       lea    0x0(%esi),%esi
```

This generalizes to:

```
; FOR
L1:
    cmp    ?, ?
    jcc    L2
    jmp     L3
L2:
    ?    ?, ?        ; body of FOR loop
    inc    ?
    jmp    L1
; ENDFOR
L3:
```

which demonstrates that the FOR statement is really an instance of a WHILE statement, albeit often with an `inc` or a `dec` at the tail of L2.

The IF-ELSE statement is generally a series of conditional and unconditional jumps that skip blocks of code. The typical model is to follow a condition test with a conditional jump that skips the next block of code; that block of code then ends with an unconditional jump that exits the IF-ELSE block. This is how gcc handles the IF-ELSE. A simple IF statement in C, such as:

```
if ( argc > 4 ) { x++; }
```

compiles to the following under gcc:

```
80483e0:       83 7d 08 04             cmpl   $0x4,0x8(%ebp)
80483e4:       7e 03                   jle    80483e9 <main+0x19>
80483e6:       ff 45 fc                incl   0xfffffffc(%ebp)
```

The generalization of this code is:

```
; IF
    cmp    ?, ?
    jcc    L1    ; jump over instructions
    ?    ?, ?    ; body of IF statement
; ENDIF
L1:
```

A more complex IF statement with an ELSE clause in C such as:

```
if ( argc > 4 ) { x++; } else { y--; }
```

compiles to the following under gcc:

```
80483e0:       83 7d 08 04             cmpl   $0x4,0x8(%ebp)
80483e4:       7e 0a                   jle    80483f0 <main+0x20>
80483e6:       ff 45 fc                incl   0xfffffffc(%ebp)
80483e9:       eb 08                   jmp    80483f3 <main+0x23>
80483eb:       90                      nop
80483ec:       8d 74 26 00             lea    0x0(%esi,1),%esi
80483f0: ff 4d f8 decl 0xfffffff8(%ebp)
```

The generalization of the IF-ELSE is therefore:

```
; IF
    cmp    ?, ?
    jcc    L1        ; jump to else condition
    ?    ?, ?        ; body of IF statement
    jmp    L2        ; jump over else
; ELSE
L1:
    ?    ?, ?        ; body of ELSE statement
; ENDIF
L2:
```

The final form of the IF contains an ELSE-IF clause:

```
if (argc > 4) {x++;} else if (argc < 24) {x *= y;} else {y--;}
```

This compiles to:

```
80483e0:       83 7d 08 04            cmpl   $0x4,0x8(%ebp)
80483e4:       7e 0a                  jle    80483f0 <main+0x20>
80483e6:       ff 45 fc               incl   0xfffffffc(%ebp)
80483e9:       eb 1a                  jmp    8048405 <main+0x35>
80483eb:       90                     nop
80483ec:       8d 74 26 00            lea    0x0(%esi,1),%esi
80483f0:       83 7d 08 17            cmpl   $0x17,0x8(%ebp)
80483f4:       7f 0c                  jg     8048402 <main+0x32>
80483f6:       8b 45 fc               mov    0xfffffffc(%ebp),%eax
80483f9:       0f af 45 f8            imul   0xfffffff8(%ebp),%eax
80483fd:       89 45 fc               mov    %eax,0xfffffffc(%ebp)
8048400:       eb 03                  jmp    8048405 <main+0x35>
8048402:       ff 4d f8               decl   0xfffffff8(%ebp)
```

The generalization of this construct is therefore:

```
; IF
    cmp    ?, ?
    jcc    L1        ; jump to ELSE-IF
    ?    ?, ?        ; body of IF statement
    jmp    L3        ; jump out of IF statement
; ELSE IF
L1:
    cmp    ?, ?
    jcc    L2        ; jump to ELSE
    ?    ?, ?        ; body of ELSE-IF statement
    jmp    L3
; ELSE
L2:
    ?    ?, ?        ; body of ELSE statement
; ENDIF
L3:
```

An alternative form of the IF will have the conditional jump lead into the code block and be followed immediately by an unconditional jump that skips the code block. This results in more jump statements but causes the condition to be identical with that of the C code (note that in the example above, the condition must be inverted so that the conditional branch will skip the code block associated with the IF).

Note that most SWITCH statements will look like IF-ELSEIF statements; large SWITCH statements will often be compiled as jump tables.

The generalized forms of the above constructs can be recognized using scripts to analyze the intermediate code produced in the previous section. For example, the IF-ELSE construct:

```
cmp    ?, ?
    jcc    L1        ; jump to else condition
    jmp    L2        ; jump over else
L1:
L2:
```

would be recognized by the following code:

```
if ( $$insn{type} == "INSN_CMP" &&
         ${$$insn{next}}{type} == "INSN_BRANCHCC" ) {
    $else_insn = get_insn_by_addr( ${$$insn{next}}{dest} );
    if ( ${$$else_insn{prev}}{type} == "INSN_BRANCH" ) {
        # This is an IF/ELSE
        $endif_insn = get_insn_by_addr( ${$$else_insn{prev}}{dest} );
        insert_before( $insn, "IF" );
        insert_before( ${$$insn{next}}{next}, "{" );
        insert_before( $else_insn, "}" );
        insert_before( $else_insn, "ELSE" );
        insert_before( $else_insn, "{" );
        insert_before( $endif_insn, "}" );
    }
}
```

The `insert_before` routine adds a pseudoinstruction to the linked list of disassembled instructions, so that the disassembled IF-ELSE in the previous section prints out as:

```
IF
80483e0:       83 7d 08 04             cmpl   $0x4,0x8(%ebp)
80483e4:       7e 0a                   jle    80483f0 <main+0x20>
{
80483e6:       ff 45 fc                incl   0xfffffffc(%ebp)
80483e9:       eb 08                   jmp    80483f3 <main+0x23>
80483eb:       90                      nop
80483ec:       8d 74 26 00             lea    0x0(%esi,1),%esi
} ELSE {
80483f0:       ff 4d f8                decl   0xfffffff8(%ebp)
}
```

By creating scripts that generate such output, supplemented perhaps by an analysis of the conditional expression to a flow control construct, the output of a disassembler can be brought closer to the original high-level language source code from which it was compiled.

Problem Areas

So far, the reverse engineering process that has been presented is an idealized one; all tools are assumed to work correctly on all targets, and the resulting disassembly is assumed to be accurate.

In most real-world reverse engineering cases, however, this is not the case. The tools may not process the target at all, or may provide an inaccurate disassembly of the underlying machine code. The target may contain hostile code, be encrypted or compressed, or simply have been compiled using nonstandard tools.

The purpose of this section is to introduce a few of the common difficulties encountered when using these tools. It's not an exhaustive survey of protection techniques, nor does it pretend to provide reasonable solutions in all cases; what follows should be considered background for the next section of this chapter, which discusses the writing of new tools to compensate for the problems the current tools cannot cope with.

Antidebugging

The prevalence of open source software on Linux has hampered the development of debuggers and other binary analysis tools; the developers of debuggers still rely on ptrace, a kernel-level debugging facility that is intended for working with "friendly" programs. As has been more than adequately shown (see the "References" section for more information), ptrace cannot be relied on for dealing with foreign or hostile binaries.

The following simple—and by now, quite common—program locks up when being debugged by a ptrace-based debugger:

```
#include <sys/ptrace.h>
    #include <stdio.h>
    int main( int argc, char **argv ) {
        if ( ptrace(PTRACE_TRACEME, 0, NULL, NULL) < 0 ) {
            /* we are being debugged */
            while (1) ;
        }
        printf("Success: PTRACE_TRACEME works\n");
        return(0);
    }
```

On applications that tend to be less obvious about their approach, the call to ptrace will be replaced with an int 80 system call:

```
asm("\t xorl %ebx, %ebx    \n"    /* PTRACE_TRACEME = 0 */
    "\t movl $26, %ea    \n"    /* from /usr/include/asm.unistd.h */
    "\t int 80        \n"    /* system call trap */
    );
```

These work because ptrace checks the task struct of the caller and returns -1 if the caller is currently being ptrace( )ed by another process. The check is very simple, but is done in kernel land:

```
/* from /usr/src/linux/arch/i386/kernel/ptrace.c */
if (request == PTRACE_TRACEME) {
        /* are we already being traced? */
        if (current->ptrace & PT_PTRACED)
              goto out;
        /* set the ptrace bit in the process flags. */
        current->ptrace |= PT_PTRACED;
        ret = 0;
        goto out;
  }
```

The usual response to this trick is to jump over or NOP out the call to ptrace, or to change the condition code on the jump that checks the return value. A more graceful way—and this extends beyond ptrace as a means of properly dealing with system calls in the target—is to simply wrap ptrace with a kernel module:

```
/*-------------------------------------------------------------------------*/
    /* ptrace wrapper: compile with `gcc -c new_ptrace.c`
                      load with    `insmod -f new_ptrace.o`
                      unload with  `rmmod new_ptrace`        */
    #define __KERNEL__
    #define MODULE
    #define LINUX

    #include <linux/kernel.h>  /* req */
    #include <linux/module.h>  /* req */
    #include <linux/init.h>    /* req */
    #include <linux/unistd.h>  /* syscall table */
    #include <linux/sched.h>   /* task struct, current( ) */
    #include <linux/ptrace.h>  /* for the ptrace types */


    asmlinkage int (*old_ptrace)(long req, long pid, long addr, long data);

    extern long sys_call_table[];

    asmlinkage int new_ptrace(long req, long pid, long addr, long data){
        /* if the caller is currently being ptrace( )ed: */
           if ( current->ptrace & PT_PTRACED ) {
              if ( req == PTRACE_TRACEME ||
                        req == PTRACE_ATTACH  ||
                        req == PTRACE_DETACH  ||
                 req == PTRACE_CONT      )
                /* lie to it and say everything's fine */
                    return(0);

              /* notify user that some other ptrace was encountered */
              printk("Prevented pid %d (%s) from ptrace(%ld) on %ld\n",
                       current->pid, current->comm, request, pid );
```

```
            return(-EIO); /* the standard ptrace( ) ret val */
         }

         return((*old_ptrace)(req, pid, addr, data));
    }

    int __init init_new_ptrace(void){
         EXPORT_NO_SYMBOLS;
       /* save old ptrace system call entry, replace it with ours */
         old_ptrace = (int(*)(long request, long pid, long addr,
              long data))   (sys_call_table[__NR_ptrace]);
         sys_call_table[__NR_ptrace] = (unsigned long) new_ptrace;
         return(0);
    }

    void __exit exit_new_ptrace(void){
       /* put the original syscall entry back in the syscall table */
         if ( sys_call_table[__NR_ptrace] != (unsigned long) new_ptrace )
            printk("Warning: someone hooked ptrace( ) after us. "
               "Reverting.\n");
         sys_call_table[__NR_ptrace] = (unsigned long) old_ptrace;
         return;
    }

    module_init(init_new_ptrace);       /* export the init routine */
    module_exit(exit_new_ptrace);       /* export the exit routine */
/*-----------------------------------------------------------------------*/
```

This is, of course, a small taste of what can be done in kernel modules; between hooking system calls and redirecting interrupt vectors (see the "References" section for more on these), the reverse engineer can create powerful tools with which to examine and monitor hostile programs.

Many automated debugging or tracing tools are based on ptrace and, as a result, routines such the following have come into use:

```
/* cause a SIGTRAP and see if it gets through the debugger */
    int being_debugged = 1;
    void int3_count( int signum ) {
        being_debugged = 0;
    }
    int main( int argc, char **argv ) {
        signal(SIGTRAP, int3_count);
        asm( "\t int3 \n");
        /* ... */
        if ( being_debugged ) {
            while (1) ;
        }
        return(0);
    }
```

With a live debugger such as gdb, these pose no problem: simply sending the generated signal to the process with gdb's `signal SIGTRAP` command fools the process into thinking it has received the signal without interference. In order to make the target work with automatic tracers, the signal specified in the `signal` call simply has to be changed to a user signal:

```
68 00 85 04 08          push   $0x8048500
     6a 05                  push   $0x5           ; SIGTRAP
     e8 83 fe ff ff         call   80483b8 <_init+0x68>

    ... becomes ...

    68 00 85 04 08          push   $0x8048500
       6a 05                  push   $0x1E           ; SIGUSR1
        e8 83 fe ff ff         call   80483b8 <_init+0x68>
```

A final technique that is fairly effective is to scan for embedded debug trap instructions (`int3` or `0xCC`) in critical sections of code:

```
/* we need the extern since C cannot see into the asm statement */
extern void here(void);
int main( int argc, char **argv ) {
    /* check for a breakpoint at the code label */
      if ( *(unsigned char *)here == 0xCC ) {
        /* we are being debugged */
            return(1);
      }
    /* create code label with an asm statement */
    asm("\t here: \n\t nop \n");
    printf("Not being debugged\n");
      return(0);
}
```

In truth, this only works because gdb's support for debug registers DR0–DR3 via its `hbreak` command is broken. Since the use of the debug registers is supported by ptrace (see the "Debugging with ptrace" section later in this chapter), this is most likely a bug or forgotten feature; however, GNU developers are nothing if not inscrutable, and it may be up to alternative debuggers such as ald or ups to provide adequate debug register support.

Antidisassembly

The name of this section is somewhat a misnomer. Typical antidisassembler techniques such as the "off-by-one-byte" and "false return" tricks will not be discussed here; by and large, such techniques fool disassemblers but fail to stand up to a few minutes of human analysis and can be bypassed with an interactive disassembler or by restarting disassembly from a new offset. Instead, what follows is a discussion of mundane problems that are much more likely to occur in practice and can be quite tedious, if not difficult, to resolve.

One of the most common techniques to obfuscate a disassembly is static linking. While this is not always intended as obfuscation, it does frustrate the analysis of the target, since library calls are not easily identified. In order to resolve this issue, a disassembler or other analysis tool that matches signatures for functions in a library (usually libc) with sequences of bytes in the target.

The technique for generating a file of signatures for a library is to obtain the exported functions in the library from the file header (usually an AR file, as documented in */usr/include/ar.h*), then iterate through the list of functions, generating a signature of no more than SIGNATURE_MAX bytes for all functions that are SIGNATURE_MIN lengths or greater in length. The values of these two constants can be obtained by experimentation; typical values are 128 bytes and 16 bytes, respectively.

Generating a function signature requires disassembling up to SIGNATURE_MAX bytes of an instruction, halting the disassembly when an unconditional branch (`jmp`) or return (`ret`) is encountered. The disassembler must be able to mask out variant bytes in an instruction with a special wildcard byte; since 0xF1 is an invalid opcode in the Intel ISA, it makes an ideal wildcard byte.

Determining which bytes are invariant requires special support that most disassemblers do not have. The goal is to determine which bytes in an instruction do not change—in general, the opcode, ModR/M byte, and SIB byte will not change. More accurate information can be found by examining the Intel Opcode Map (see the "References" section for more information); the addressing methods of operands give clues as to what may or may not change during linking:

```
* Methods C D F G J P S T V X Y are always invariant
* Methods E M Q R W contain ModR/M and SIB bytes which may contain
  variant bytes, according to the following conditions:
    If the ModR/M 'mod' field is 00 and either 1) the ModR/M 'rm'
    field is 101 or 2) the SIB base field is 101, then the 16- or
    32-bit displacement of the operand is variant.
* Methods I J are variant if the type is 'v' [e.g., Iv or Jv]
* Methods A O are always variant
```

The goal of signature generation is to create as large a signature as possible, in which all of the variant (or prone to change in the linking process) bytes are replaced with wildcard bytes.

When matching library function signatures to byte sequences in a binary, a byte-for-byte comparison is made, with the wildcard bytes in the signature always matching bytes in the target. If all of the bytes in the signature match those in the target, a label is created at the start of the matching byte sequence that bears the name of the library function. Note that it is important to implement this process so that as few false positives are produced as possible; this means signature collisions—i.e., two library functions with identical signatures—must be resolved by discarding both signatures.

One of the greatest drawbacks of the GNU binutils package (the collection of tools containing ld, objdump, objcopy, etc.) is that its tools are entirely unable to handle binaries that have had their ELF section headers removed (see the upcoming section "The ELF File Format"). This is a serious problem, for two reasons: first of all, the Linux ELF loader will load and execute anything that has ELF program headers but, in accordance with the ELF standard, it assumes the section headers are optional; and secondly, the ELF Kickers (see the "References" section) package contains a utility called sstrip that removes extraneous symbols and ELF section headers from a binary.

The typical approach to an sstriped binary is to switch tools and use a disassembler without these limitations, such as IDA, ndisasm, or even the embedded disassembler in biew or hte. This is not really a solution, though; currently, there are tools in development or in private release that attempt to rebuild the section headers based on information in the program headers.

Writing New Tools

As seen in the previous section, the current tools based on binutils and ptrace leave a lot to be desired. While there are currently tools in development that compensate for these shortcomings, the general nature of this book and the volatile state of many of the projects precludes mentioning them here. Instead, what follows is a discussion of the facilities available for writing new tools to manipulate binary files.

The last half of this chapter contains a great deal of example source code. The reader is assumed to be familiar with C as well as with the general operation of binary tools such as linkers, debuggers, and disassemblers. This section begins with a discussion of parsing the ELF file header, followed by an introduction to writing programs using ptrace(2) and a brief look at the GNU BFD library. It ends with a discussion of using GNU libopcodes to create a disassembler.

The ELF File Format

The standard binary format for Linux and Unix executables is the Executable and Linkable Format (ELF). Documentation for the ELF format is easily obtainable; Intel provides PDF documentation at no charge as part of its Tool Interface Standards series (see the "References" section at the end of this chapter for more information).

Typical file types in ELF include binary executables, shared libraries, and the object or ".o" files produced during compilation. Static libraries, or ".a" files, consist of a collection of ELF object files linked by AR archive structures.

An ELF file is easily identified by examining the first four bytes of the file; they must be \177ELF, or 7F 45 4C 46 in hexdecimal. This four-byte signature is the start of the ELF file header, which is defined in */usr/include/elf.h*:

```
typedef struct {                        /* ELF File Header */
    unsigned char   e_ident[16];        /* Magic number */
    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 addr */
    Elf32_Off       e_phoff;            /* Prog hdr tbl file offset */
    Elf32_Off       e_shoff;            /* Sect hdr tbl file offset */
    Elf32_Word      e_flags;            /* Processor-specific flags */
    Elf32_Half      e_ehsize;           /* ELF header size in bytes */
    Elf32_Half      e_phentsize;        /* Prog hdr tbl entry size */
    Elf32_Half      e_phnum;            /* Prog hdr tbl entry count */
    Elf32_Half      e_shentsize;        /* Sect hdr tbl entry size */
    Elf32_Half      e_shnum;            /* Sect hdr tbl entry count */
    Elf32_Half      e_shstrndx;         /* Sect hdr string tbl idx */
} Elf32_Ehdr;
```

Following the ELF header are a table of section headers and a table of program headers; the section headers represent information of interest to a compiler tool suite, while program headers represent everything that is needed to link and load the program at runtime. The difference between the two header tables is the cause of much confusion, as both sets of headers refer to the same code or data in the program.

Program headers are required for the program to run; each header in the table refers to a segment of the program. A segment is a series of bytes with one of the following types associated with it:

```
PT_LOAD         -- Bytes that are mapped as part of the process image
    PT_DYNAMIC  -- Information passed to the dynamic linker
    PT_INTERP   -- Path to interpreter, usually "/lib/ld-linux.so.2"
    PT_NOTE     -- Vendor-specific information
    PT_PHDR     -- This segment is the program header table
```

Each program header has the following structure:

```
typedef struct {                        /* ELF Program Segment Header */
    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;
```

Note that each program segment has a file offset as well as a virtual address, which is the address that the segment expects to be loaded into at runtime. The segments also have both "in-file" and "in-memory" sizes: the "in-file" size specifies how many bytes to read from the file, and "in-memory" specifies how much memory to allocate for the segment.

In contrast, the section headers have the following structure:

```
typedef struct {
        Elf32_Word      sh_name;          /* Section name */
        Elf32_Word      sh_type;          /* Section type */
```

```
        Elf32_Word      sh_flags;        /* Section flags */
        Elf32_Addr      sh_addr;         /* Section virtual addr */
        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 info */
        Elf32_Word      sh_addralign;    /* Section alignment */
        Elf32_Word      sh_entsize;      /* Section table entry size */
    } Elf32_Shdr;
```

Sections have the following types:

```
SHT_PROGBITS      -- Section is mapped into process image
SHT_SYMTAB        -- Section is a Symbol Table
SHT_STRTAB        -- Section is a String Table
SHT_RELA          -- Section holds relocation info
SHT_HASH          -- Section is a symbol hash table
SHT_DYNAMIC       -- Section contains dynamic linking info
SHT_NOTE          -- Section contains vendor-specific info
SHT_NOBITS        -- Section is empty but is mapped, e.g., ".bss"
SHT_REL           -- Section holds relocation info
SHT_DYNSYM        -- Section contains Dynamic Symbol Table
```

As noted, sections are redundant with program segments and often refer to the same bytes in the file. It is important to realize that sections are not mandatory and may be removed from a compiled program by utilities such as sstrip. One of the greatest failings of the GNU binutils tools is their inability to work with programs that have had their section headers removed.

For this reason, only program segment headers will be discussed; in fact, all that is needed to understand the file structure are the program headers, the dynamic string table, and the dynamic symbol table. The PT_DYNAMIC segment is used to find these last two tables; it consists of a table of dynamic info structures:

```
typedef struct {                  /* ELF Dynamic Linking Info */
    Elf32_Sword     d_tag;        /* Dynamic entry type */
    union {
        Elf32_Word d_val;         /* Integer value */
        Elf32_Addr d_ptr;         /* Address value */
    } d_un;
} Elf32_Dyn;
```

The `dt_tag` field specifies the type of information that is pointed to by the `d_val` or `d_ptr` fields; it has many possible values, with the following being those of greatest interest:

```
DT_NEEDED    -- String naming a shared library needed by the program
DT_STRTAB    -- Virtual Address of the Dynamic String Table
DT_SYMTAB    -- Virtual Address of the Dynamic Symbol Table
DT_STRSZ     -- Size of the Dynamic String Table
DT_SYMENT    -- Size of a Dynamic Symbol Table element
DT_INIT      -- Virtual Addr of an initialization (".init") function
DT_FINI      -- Virtual Addr of a termination (".fini") function
DT_RPATH     -- String giving a path to search for shared libraries
```

It should be noted that any information that consists of a string actually contains an index in the dynamic string table, which itself is simply a table of NULL-terminated strings; referencing the dynamic string table plus the index provides a standard C-style string. The dynamic symbol table is a table of symbol structures:

```
typedef struct {                        /* ELF Symbol */
    Elf32_Word      st_name;            /* Symbol name (strtab index) */
    Elf32_Addr      st_value;           /* Symbol value */
    Elf32_Word      st_size;            /* Symbol size */
    unsigned char   st_info;            /* Symbol type and binding */
    unsigned char   st_other;           /* Symbol visibility */
    Elf32_Section   st_shndx;           /* Section index */
} Elf32_Sym;
```

Both the string and symbol tables are for the benefit of the dynamic linker and they contain no strings or symbols associated with the source code of the program.

By way of disclaimer, it should be noted that this description of the ELF format is minimal and is intended only for understanding the section that follows. For a complete description of the ELF format, including sections, the PLT and GOT, and issues such as relocation, see the Intel specification.

Sample ELF reader

The following source code demonstrates how to work with the ELF file format, since the process is not immediately obvious from the documentation. In this routine, "buf" is assumed to be a pointer to a memory-mapped image of the target, and "buf_len" is the length of the target.

```
/*---------------------------------------------------------------------------*/
    #include <elf.h>

    unsigned long elf_header_read( unsigned char *buf, int buf_len ){
        Elf32_Ehdr *ehdr = (Elf32_Ehdr *)buf;
        Elf32_Phdr *ptbl = NULL, *phdr;
        Elf32_Dyn  *dtbl = NULL, *dyn;
        Elf32_Sym  *symtab = NULL, *sym;
        char       *strtab = NULL, *str;
        int         i, j, str_sz, sym_ent, size;
        unsigned long offset, va;    /* file pos, virtual address */
        unsigned long entry_offset;    /* file offset of entry point */

        /* set the default entry point offset */
        entry_offset =  ehdr->e_entry;

        /* iterate over the program segment header table */
        ptbl = (Elf32_Phdr *)(buf + ehdr->e_phoff);

        for ( i = 0; i < ehdr->e_phnum; i++ ) {
            phdr = &ptbl[i];

            if ( phdr->p_type == PT_LOAD ) {
```

```
        /* Loadable segment: program code or data */
        offset = phdr->p_offset;
        va = phdr->p_vaddr;
        size = phdr->p_filesz;

        if ( phdr->p_flags & PF_X ) {
            /* this is a code section */
        } else if ( phdr->p_flags & (PF_R | PF_W) ){
            /* this is read/write data */
        } else if (phdr->p_flags & PF_R ) {
            /* this is read-only data */
        }    /* ignore other sections */

        /* check if this contains the entry point */
        if ( va <= ehdr->e_entry &&
             (va + size) > ehdr->e_entry ) {
            entry_offset = offset + (entry - va);
        }

    } else if ( phdr->p_type == PT_DYNAMIC ) {
        /* dynamic linking info: imported routines */
        dtbl = (Elf32_Dyn *) (buf + phdr->p_offset);

        for ( j = 0; j < (phdr->p_filesz /
                sizeof(Elf32_Dyn)); j++ ) {
            dyn = &dtbl[j];
            switch ( dyn->d_tag ) {
            case DT_STRTAB:
                strtab = (char *)
                    dyn->d_un.d_ptr;
                break;
            case DT_STRSZ:
                str_sz = dyn->d_un.d_val;
                break;
            case DT_SYMTAB:
                symtab = (Elf32_Sym *)
                    dyn->d_un.d_ptr;
                break;
            case DT_SYMENT:
                sym_ent = dyn->d_un.d_val;
                break;
            case DT_NEEDED:
                /* dyn->d_un.d_val is index of
                   library name in strtab */
                break;
            }
        }

    }    /* ignore other program headers */
}

/* make second pass looking for symtab and strtab */
for ( i = 0; i < ehdr->e_phnum; i++ ) {
    phdr = &ptbl[i];
```

```
            if ( phdr->p_type == PT_LOAD ) {
                if ( strtab >= phdr->p_vaddr && strtab <
                    phdr->p_vaddr + phdr->p_filesz ) {
                    strtab = buf + phdr->p_offset +
                        ((int) strtab - phdr->p_vaddr);
                }
                if ( symtab >= phdr->p_vaddr && symtab <
                            phdr->p_vaddr +
                            phdr->p_filesz ) {
                    symtab = buf + phdr->p_offset +
                        ((int) symtab - phdr->p_vaddr);
                }
            }
        }

        if ( ! symtab )    {
            fprintf(stderr, "no symtab!\n");
            return(0);
        }
        if ( ! strtab )    {
            fprintf(stderr, "no strtab!\n");
            return(0);
        }
        /* handle symbols for functions and shared library routines */
        size = strtab - (char *)symtab;    /* strtab follows symtab */

        for ( i = 0; i < size / sym_ent; i++ ) {
            sym = &symtab[i];
            str = &strtab[sym->st_name];

            if ( ELF32_ST_TYPE( sym->st_info ) == STT_FUNC ){
                /* this symbol is the name of a function */
                offset = sym->st_value;

                if ( sym->st_shndx ) {
                /* 'str' == subroutine at 'offset' in file */
                    ;
                } else {
                /* 'str' == name of imported func at 'offset' */
                    ;
                }
            }    /* ignore other symbols */
        }

        /* return the entry point */
        return( entry_offset );
    }
/*---------------------------------------------------------------------*/
```

A few notes are needed to clarify the source code. First, the locations of the string and symbol tables are not immediately obvious; the dynamic info structure provides their virtual addresses, but not their locations in the file. A second pass over the program headers is used to find the segment containing each so that their file offsets can be determined; in a real application, each segment will have been added to a list for future processing, so the second pass will be replaced with a list traversal.

The length of the symbol table is also not easy to determine; while it could be found by examining the section headers, in practice it is known that GNU linkers place the string table immediately after the symbol table. It goes without saying that a real application should use a more robust method.

Note that section headers can be handled in the same manner as the program headers, using code such as:

```
Elf32_Shdr *stbl, *shdr;

stbl = buf + ehdr->s_shoff;    /* section header table */
for ( i = 0; i < ehdr->e_shnum; i++ ) {
    shdr = &stbl[i];

    switch ( shdr->sh_type ) {
        /* ... handle different section types here */
    }
}
```

The symbol and string tables in the section headers use the same structure as those in the program headers.

Here is the code used for loading a target when implementing the above ELF routines:

```
/*---------------------------------------------------------------------------*/
    #include <errno.h>
    #include <fcntl.h>
    #include <stdio.h>
    #include <sys/mman.h>
    #include <sys/stat.h>
    #include <sys/types.h>
    #include <unistd.h>

    int main( int argc, char **argv ) {
        int fd;
        unsigned char *image;
        struct stat s;

        if ( argc < 2 ) {
            fprintf(stderr, "Usage: %s filename\n", argv[0]);
            return(1);
        }
        if ( stat( argv[1], &s) ) {
            fprintf(stderr, "Error: %s\n", strerror(errno) );
            return(2);
        }
        fd = open( argv[1], O_RDONLY );
        if ( fd < 0 )  {
            fprintf(stderr, "Error: %s\n", strerror(errno) );
            return(3);
        }
```

```
        image = mmap(0, s.st_size, PROT_READ, MAP_SHARED, fd, 0);
        if ( (int) image < 0 ) {
            fprintf(stderr, "Error: %s\n", strerror(errno) );
            return(4);
        }

        /* at this point the file can be accessed via 'fd' or 'image' */
        printf( "Offset of entry point: 0x%X\n",
                    elf_header_read( image, s.st_size ) );

        munmap( image, s.st_size );
        close(fd);
        return(0);
    }
/*-------------------------------------------------------------------*/
```

Debugging with ptrace

On Unix and Linux (or, to split a further hair, GNU/Linux) systems, process debugging is provided by the kernel ptrace(2) facility. The purpose of ptrace is to allow one process to access and control another; this means that ptrace provides routines to read and write to the memory of the target process, to view and set the registers of the target process, and to intercept signals sent to the target.

This last feature is perhaps the most important, though it is often left unstated. On the Intel architecture, debug traps (i.e., traps caused by breakpoints) and trace traps (caused by single-stepping through code) raise specific interrupts: interrupts 1 and 3 for debug traps, and interrupt 1 for trace traps. The interrupt handlers in the kernel create signals that are sent to the process in whose context the trap occurred. Debugging a process is therefore a matter of intercepting these signals before they reach the target process and analyzing or modifying the state of the target based on the cause of the trap.

The ptrace API is based around this model of intercepting signals sent to the target:

```
/* attach to process # pid */
        int pid, status, cont = 1;

          if ( ptrace( PTRACE_ATTACH, pid, 0, 0) == -1 ) {
            /* failed to attach: do something terrible */
        }

        /* if PTRACE_ATTACH succeeded, target is stopped */
        while ( cont && err != -1 )
            /* target is stopped -- do something */
            /* PTRACE_?? is any of the ptrace routines */
                err = ptrace( PTRACE_CONT, pid, NULL, NULL);
            /* deal with result of ptrace( ) */

            /* continue execution of the target */
                err = ptrace( PTRACE_CONT, pid, NULL, NULL);
```

```
            wait(&status);

            /* target has stopped after the CONT */
               if ( WIFSIGNALED(status) ) {
                /* handle signal in WTERMSIG(status) */
            }
        }
```

Here the debugger receives control of the target in two cases: when the target is initially attached to and when the target receives a signal. As can be seen, the target will only receive a signal while it is executing—i.e., after being activated with the PTRACE_CONT function. When a signal has been received, the wait(2) returns and the debugger can examine the target. There is no need to send a SIGSTOP, as ptrace has taken care of this.

The following functions are provided by ptrace:

```
PTRACE_ATTACH     -- attach to a process [SIGSTOP]
PTRACE_DETACH     -- detach from a ptraced process [SIGCONT]
PTRACE_TRACEME    -- allow parent to ptrace this process [SIGSTOP]
PTRACE_CONT       -- Continue a ptraced process [SIGCONT]
PTRACE_KILL       -- Kill the process [sends SIGKILL]
PTRACE_SINGLESTEP -- Execute one instruction of a ptraced process
PTRACE_SYSCALL    -- Execute until entry/exit of syscall [SIGCONT, SIGSTOP]
PTRACE_PEEKTEXT   -- get data from .text segmen of ptraced processt
PTRACE_PEEKDATA   -- get data from .data segmen of ptraced processt
PTRACE_PEEKUSER   -- get data from kernel user struct of traced process
PTRACE_POKETEXT   -- write data to .text segment of ptraced process
PTRACE_POKEDATA   -- write data to .data segment of ptraced process
PTRACE_POKEUSER   -- write data from kernel user struct of ptraced process
PTRACE_GETREGS    -- Get CPU registers of ptraced process
PTRACE_SETREGS    -- Set CPU registers of ptraced process
PTRACE_GETFPREGS  -- Get floating point registers of ptraced process
PTRACE_SETFPREGS  -- Set floating point registers of ptraced process
```

Implementing standard debugger features with these functions can be complex; ptrace is designed as a set of primitives upon which a debugging API can be built, but it is not itself a full-featured debugging API.

Consider the case of tracing or single-stepping a target. The debugger first sets the TF flag (0x100) in the eflags register of the target, then starts or continues the execution of the target. The INT1 generated by the trace flag sends a SIGTRAP to the target; the debugger intercepts it, verifies that the trap is caused by a trace and not by a breakpoint (usually by looking at the debug status register DR6 and examining the byte at eip to see if it contains an embedded INT3), and sends a SIGSTOP to the target. At this point, the debugger allows the user to examine the target and choose the next action; if the user chooses to single-step the target again, the TF flag is set again (the CPU resets TF after a single instruction has executed) and a SIGCONT is sent to the target; otherwise, if the user chooses to continue execution of the target, just the SIGCONT is sent.

The ptrace facility performs much of this work itself; it provides functions that single-step a target:

```
     err = ptrace( PTRACE_SINGLESTEP, pid, NULL, NULL);
wait(&status);
if ( WIFSIGNALED(status) && WTERMSIG(status) == SIGTRAP ) {
    /* we can assume this is a single-step if we
       have set no BPs, or we can examine DR6 to
       be sure ... see coverage of debug registers */
}
```

on return from the wait(2), the target executed a single instruction and was stopped; subsequent calls to `ptrace(PTRACE_SINGLESTEP)` will step additional instructions.

The case of a breakpoint is slightly different. Here, the debugger installs a breakpoint either by setting a CPU debug register or by embedding a debug trap instruction (INT3) at the desired code address. The debugger then starts or continues execution of the target and waits for a SIGTRAP. This signal is intercepted, the breakpoint disabled, and the instruction executed. Note that this process can be quite intricate when using embedded trap instructions; the debugger must replace the trap instruction with the original byte at that address, decrement the instruction pointer (the eip register) in order to re-execute the instruction that contained the embedded debug trap, single-step an instruction, and re-enable the breakpoint.

In ptrace, an embedded or hardware breakpoint is implemented as follows:

```
unsigned long old_insn, new_insn;
old_insn = ptrace( PTRACE_PEEKTEXT, pid, addr, NULL );
if ( old_insn != -1 ) {
    new_insn = old_insn;
    ((char *)&new_insn)[0] = 0xCC;    /* replace with int3 */
    err = ptrace( PTRACE_POKETEXT, pid, addr, &new_insn );
    err = ptrace( PTRACE_CONT, pid, NULL, NULL );
    wait(&status);
    if ( WIFSIGNALED(status) && WTERMSIG(status) == SIGTRAP ) {
        /* check that this is our breakpoint */
        err = ptrace( PTRACE_GETREGS, pid, NULL, &regs);
        if ( regs.eip == addr ) {
            /* -- give user control before continue -- */
            /* disable breakpoint ... */
            err = ptrace( PTRACE_POKETEXT, pid, addr,
                    &old_insn );
            /* execute the breakpointed insn ... */
                 err = ptrace( PTRACE_SINGLESTEP, pid, NULL,
                    NULL );
            /* re-enable the breakpoint */
            err = ptrace( PTRACE_POKETEXT, pid, addr,
                    &new_insn );
        }
    }
}
```

As can be seen, ptrace does not provide any direct support for breakpoints; however, support for breakpoints can be written quite easily.

Despite the fact that widely used ptrace-based debuggers do not implement breakpoints using Intel debug registers, ptrace itself provides facilities for manipulating these registers. The support for this can be found in the sys_ptrace routine in the Linux kernel:

```
/* defined in /usr/src/linux/include/linux/sched.h */
    struct task_struct {
        /* ... */
        struct user_struct *user;
        /* ... */
    };

    /* defined in /usr/include/sys/user.h */
    struct user {
        struct user_regs_struct regs;
        /* ... */
        int u_debugreg[8];
    };

    /* from /usr/src/linux/arch/i386/kernel/ptrace.c */
    int sys_ptrace(long request, long pid, long addr, long data) {
          struct task_struct *child;
          struct user * dummy = NULL;

        /* ... */

          case PTRACE_PEEKUSR:
        unsigned long tmp;
        /* ... check that address is in struct user ...  */
        /* ... hand off reading of normal regs to getreg( ) ... */

        /* if address is a valid debug register: */
                if(addr >= (long) &dummy->u_debugreg[0] &&
                   addr <= (long) &dummy->u_debugreg[7]){
                      addr -= (long) &dummy->u_debugreg[0];
                      addr = addr >> 2;
                      tmp = child->thread.debugreg[addr];
                }
            /* write contents using put_user( ) */
                break;
        /* ... */

    case PTRACE_POKEUSR:
        /* ... check that address is in struct user ...  */
        /* ... hand off writing of normal regs to putreg( ) ... */

        /* if address is a valid debug register: */
                  if(addr >= (long) &dummy->u_debugreg[0] &&
                addr <= (long) &dummy->u_debugreg[7]){

            /* skip DR4 and DR5  */
            if(addr == (long) &dummy->u_debugreg[4]) break;
            if(addr == (long) &dummy->u_debugreg[5]) break;
```

```
/* do not write invalid addresses */
if(addr < (long) &dummy->u_debugreg[4] &&
    ((unsigned long) data) >= TASK_SIZE-3) break;

/* write control register DR7 */
            if(addr == (long) &dummy->u_debugreg[7]) {
                  data &= ~DR_CONTROL_RESERVED;
                  for(i=0; i<4; i++)
                        if ((0x5f54 >>
            ((data >> (16 + 4*i)) & 0xf)) & 1)
            goto out_tsk;
            }

/* write breakpoint address to DR0 - DR3 */
            addr -= (long) &dummy->u_debugreg;
            addr = addr >> 2;
            child->thread.debugreg[addr] = data;
            ret = 0;
  }
  break;
```

The debug registers exist in the user structure for each process; ptrace provides special routines for accessing data in this structure—the `PTRACE_PEEKUSER` and `PTRACE_POKEUSER` commands. These commands take an offset into the user structure as the `addr` parameter; as the above kernel excerpt shows, if the offset and data pass the validation tests, the data is written directly to the debug registers for the process. This requires some understanding of how the debug registers work.

There are eight debug registers in an Intel CPU: DR0–DR7. Of these, only the first four can be used to hold breakpoint addresses; DR4 and DR5 are reserved, DR6 contains status information following a debug trap, and DR7 is used to control the four breakpoint registers.

The DR7 register contains a series of flags with the following structure:

```
       condition word (16-31)                  control word (0-15)
  00  00  00  00  -  00  00  00  00  |  00  00  00  00  -  00  00  00  00
  Len R/W Len R/W    Len R/W Len R/W    RR  GR  RR  GL     GL  GL  GL  GL
    DR3     DR2        DR1     DR0          D       EE     33  22  11  00
```

The control word contains fields for managing breakpoints: G0–G3, Global (all tasks) Breakpoint Enable for DR0–3; L0–L3, Local (single task) Breakpoint Enable for DR0–3; GE, Global Exact breakpoint enable; LE, Local Exact breakpoint enable; and GD, General Detect of attempts to modify DR0–7.

The condition word contains a nibble for each debug register, with two bits dedicated to read/write access and two bits dedicated to data length:

```
R/W Bit    Break on...
-----------------------------------------------------
 00        Instruction execution only
 01        Data writes only
 10        I/O reads or writes
 11        Data read/write [not instruction fetches]
```

```
 Len Bit    Length of data at address
-------------------------------------------------------
 00       1 byte
 01       2 bytes
 10       Undefined
4 bytes
```

Note that data breakpoints are limited in size to the machine word size of the processor.

The following source demonstrates how to implement debug registers using ptrace. Note that no special compiler flags or libraries are needed to compile programs with ptrace support; the usual `gcc -o program_name *.c` works just fine.

```
/*---------------------------------------------------------------------------*/
    #include <errno.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/ptrace.h>
    #include <asm/user.h>     /* for struct user */

    #define MODE_ATTACH 1
    #define MODE_LAUNCH 2

    /* shorthand for accessing debug registers */
    #define DR( u, num ) u.u_debugreg[num]

    /* get offset of dr 'num' from start of user struct */
    #define DR_OFF( u, num ) (long)(&u.u_debugreg[num]) - (long)&u

    /* get DR number 'num' into struct user 'u' from procss 'pid' */
    #define GET_DR( u, num, pid )                        \
        DR(u, num) = ptrace( PTRACE_PEEKUSER, pid,         \
                  DR_OFF(u, num), NULL );

    /* set DR number 'num' to struct user 'u' from procss 'pid' */
    /* NOTE: the ptrace(2) man page is incorrect: the last argument to
         POKEUSER must be the word itself, not the address of the word
       in the parent's memory space. See arch/i386/kernel/ptrace.c  */
    #define SET_DR( u, num, pid )                        \
        ptrace( PTRACE_POKEUSER, pid, DR_OFF(u, num), DR(u, num) );

    /* return # of bytes to << in order to set/get local enable bit */
    #define LOCAL_ENABLE( num ) ( 1 << num )

    #define DR_LEN_MASK 0x3
    #define DR_LEN( num ) (16 + (4*num))

    #define DR_RWX_MASK 0x3
    #define DR_RWX( num )  (18 + (4*num))

    /* !=0 if trap is due to single step */
    #define DR_STAT_STEP( dr6 ) ( dr6 & 0x2000 )
```

```
/* !=0 if trap is due to task switch */
#define DR_STAT_TASK( dr6 ) ( dr6 & 0x4000 )

/* !=0 if trap is due to DR register access detected */
#define DR_STAT_DRPERM( dr6 ) ( dr6 & 0x8000 )

/* returns the debug register that caused the trap */
#define DR_STAT_DR( dr6 ) ( (dr6 & 0x0F)  )

/* length is 1 byte, 2 bytes, undefined, or 4 bytes */
enum dr_len { len_byte = 0, len_hword, len_unk, len_word };

/* bp condition is exec, write, I/O read/write, or data read/write */
enum dr_rwx { bp_x = 0, bp_w, bp_iorw, bp_rw };

int set_bp(int pid, unsigned long rva, enum dr_len len, enum dr_rwx rwx){
    struct user u = {0};
    int x, err, dreg = -1;

    err = errno;
    GET_DR( u, 7, pid );
    if ( err != errno ) {
        fprintf(stderr, "BP_SET read dr7 error: %s\n",
                strerror(errno));
        return(0);
    }

    /* find unused debug register */
    for ( x = 0; x < 4; x++ ){
        if ( ! DR(u, 7) & LOCAL_ENABLE( x ) ) {
            dreg = x;
            break;
        }
    }
    if ( dreg != -1 ) {
        /* set bp */
        DR(u, dreg) = rva;
        err = SET_DR( u, dreg, pid );
        if ( err == -1 ) {
            fprintf(stderr, "BP_SET DR%d error: %s\n", dreg,
                    strerror(errno));
            return;
        }
        /* enable bp and conditions in DR7 */
        DR(u, 7) &= ~(DR_LEN_MASK << DR_LEN(dreg));
        DR(u, 7) &= ~(DR_RWX_MASK << DR_RWX(dreg));

        DR(u, 7) |= len << DR_LEN(dreg);
        DR(u, 7) |= rwx << DR_RWX(dreg);
        DR(u, 7) |= LOCAL_ENABLE(dreg);
        err = SET_DR( u, 7, pid );
        if ( err == -1 ) {
            fprintf(stderr, "BP_SET DR7 error: %s\n",
                    strerror(errno));
```

```
            return;
        }
    }

    return( dreg );    /* -1 means no free debug register */
}

int unset_bp( int pid, unsigned long rva ) {
    struct user u = {0};
    int x, err, dreg = -1;

    for ( x = 0; x < 4; x++ ){
        err = errno;
        GET_DR(u, x, pid);
        if ( err != errno ) {
            fprintf(stderr, "BP_UNSET get DR%d error: %s\n", x,
                        strerror(errno));
            return(0);
        }
        if ( DR(u, x) == rva ) {
            dreg = x;
            break;
        }
    }
    if ( dreg != -1 ) {
        err = errno;
        GET_DR( u, 7, pid );
        if ( err != errno ) {
            fprintf(stderr, "BP_UNSET get DR7 error: %s\n",
                    strerror(errno));
            return(0);
        }
        DR(u, 7) &= ~(LOCAL_ENABLE(dreg));
        err = SET_DR( u, 7, pid ) ;
        if ( err == -1 ) {
            fprintf(stderr, "BP_UNSET DR7 error: %s\n",
                    strerror(errno));
            return;
        }
    }
    return(dreg);    /* -1 means no debug register set to rva */
}

/* reason for bp trap */
enum bp_status = { bp_trace, bp_task, bp_perm, bp_0, bp_1, bp_2, bp_3,
             bp_unk };

enum bp_status get_bp_status( int pid ) {
    int dreg;
    struct user u = {0};
    enum bp_status rv = bp_unk;

    GET_DR( u, 6, pid );
    printf("Child stopped for ");
```

```
        if ( DR_STAT_STEP( DR(u, 6) ) ) {
            rv = bp_trace;
        } else if  ( DR_STAT_TASK(DR(u,6)) ){
            rv = bp_task;
        } else if ( DR_STAT_DRPERM(DR(u,6)) ) {
            rv = bp_perm;
        } else {
            dreg = DR_STAT_DR(DR(u,6));
            if ( dreg == 1 ) {
                rv = bp_0;
            } else if ( dreg == 2 ) {
                rv = bp_1;
            } else if ( dreg == 4 ) {
                rv = bp_2;
            } else if ( dreg == 8 ) {
                rv = bp_3;
            }
        }
        return( rv );
    }
/*---------------------------------------------------------------------------*/
```

These routines can then be incorporated into a standard ptrace-based debugger such as the following:

```
/*---------------------------------------------------------------------------*/
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/ptrace.h>
    #include <sys/wait.h>
    #include <errno.h>
    #include <signal.h>

    #include "hware_bp.h"    /* protos for set_bp(), unset_bp( ), etc */

    #define DEBUG_SYSCALL    0x01
    #define DEBUG_TRACE    0x02

    unsigned long get_rva( char *c ) {
        unsigned long rva;
        while ( *c && ! isalnum( *c ) )
            c++;
        if ( c && *c )
            rva = strtoul( c, NULL, 16 );
        return(rva);
    }

    void print_regs( int pid ) {
        struct user_regs_struct regs;
        if (ptrace( PTRACE_GETREGS, pid, NULL, &regs) != -1 ) {
            printf("CS:IP %04X:%08X\t SS:SP %04X:%08X FLAGS %08X\n",
                regs.cs, regs.eip, regs.ss, regs.esp, regs.eflags);
            printf("EAX %08X \tEBX %08X \tECX %08X \tEDX %08X\n",
                regs.eax, regs.ebx, regs.ecx, regs.edx );
```

```
    }
    return;
}

void handle_sig( int pid, int signal, int flags ) {
    enum bp_status status;

    if ( signal == SIGTRAP ) {
        printf("Child stopped for ");

        /* see if this was caused by debug registers */
        status = get_bp_status( pid );
        if ( status == bp_trace ) {
            printf("trace\n");
        } else if  ( status == bp_task ){
            printf("task switch\n");
        } else if ( status == bp_perm ) {
            printf("attempted debug register access\n");
        } else if ( status != bp_unk ) {
            printf("hardware breakpoint\n");
        } else {
            /* nope */
            if ( flags & DEBUG_SYSCALL ) {
                printf("syscall\n");
            } else if ( flags & DEBUG_TRACE ) {
                /* this should be caught by bp_trace */
                printf("trace\n");
            }
        }

    }
    return;
}

int main( int argc, char **argv) {
      int mode, pid, status, flags = 0, err = 0, cont = 1;
    char *c, line[256];

    /* check args */
    if ( argc == 3 && argv[1][0] == '-' && argv[1][1] == 'p' ) {
        pid = strtoul( argv[2], NULL, 10 );
        mode = MODE_ATTACH;
    } else if ( argc >= 2 ) {
        mode = MODE_LAUNCH;
    } else {
        printf( "Usage: debug [-p pid] [filename] [args...]\n");
        return(-1);
    }

    /* start/attach target based on mode */

    if ( mode == MODE_ATTACH ) {
          printf("Tracing PID: %x\n", pid);
          err = ptrace( PTRACE_ATTACH, pid, 0, 0);
```

```
} else {
    if ( (pid = fork()) < 0 ) {
        fprintf(stderr, "fork() error: %s\n", strerror(errno));
        return(-2);
    } else if ( pid ) {
          printf("Executing %s PID: %x\n", argv[1], pid);
        wait(&status);

    } else {
        err = ptrace( PTRACE_TRACEME, 0, 0, 0);
        if ( err == -1 ) {
            fprintf(stderr, "TRACEME error: %s\n",
                strerror(errno));
            return(-3);
        }
        return( execv(argv[1], &argv[1]) );
    }
}

  while ( cont && err != -1 ) {
      print_regs( pid );
      printf("debug:");
      fgets( line, 256, stdin );
      for ( c = line; *c && !(isalnum(*c)) ; c++ )
          ;
        switch (*c) {
        case 'b':
            set_bp(pid, get_rva(++c), len_byte, bp_x);
            break;
        case 'r':
            unset_bp(pid, get_rva(++c));
            break;
        case 'c':
                    err = ptrace( PTRACE_CONT, pid, NULL, NULL);
            wait(&status);
            break;
        case 's':
            flags |= DEBUG_SYSCALL;
                    err = ptrace( PTRACE_SYSCALL, pid, NULL, NULL);
            wait(&status);
            break;
        case 'q':
                    err = ptrace( PTRACE_KILL, pid, NULL, NULL);
            wait(&status);
            cont = 0;
            break;
        case 't':
            flags |= DEBUG_TRACE;
                    err = ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);
            wait(&status);
            break;
        case '?':
        default:
            printf("b [addr] - set breakpoint\n"
```

```
                        "r [addr] - remove breakpoint\n"
                          "c       - continue\n"
                          "s       - run to syscall entry/exit\n"
                          "q       - kill target\n"
                          "t       - trace/single step\n" );
                   break;
           }
           if ( WIFEXITED(status) ) {
               printf("Child exited with %d\n", WEXITSTATUS(status));
               return(0);
           } else if ( WIFSIGNALED(status) ) {
               printf("Child received signal %d\n", WTERMSIG(status));
               handle_sig( pid, WTERMSIG(status), flags );
           }
       }
       if ( err == -1 )
           printf("ERROR: %s\n", strerror(errno));
         ptrace( PTRACE_DETACH, pid, 0, 0);
       wait(&status);
         return(0);
   }
/*---------------------------------------------------------------------*/
```

Naturally, for this to be a "real" debugger, it should incorporate a disassembler as well as allow the user to read and write memory addresses and registers.

The ptrace facility can also be used to monitor a running process and report on its usage of library calls, system calls, or files, or to report on its own internal state (such as signals it has received, which internal subroutines have been called, what the contents of the register were when a conditional branch was reached, and so on). Most such utilities use either PTRACE_SYSCALL or PTRACE_SINGLESTEP in order to halt the process temporarily and make a record of its activity.

The following code demonstrates the use of PTRACE_SYSCALL to record all system calls made by the target:

```
/*-----------------------------------------------------------------------*/
    struct user_regs_struct regs;
    int state = 0, err = 0, cont = 1;

      while ( cont && err != -1 ) {
        state = state ? 0 : 1;
            err = ptrace( PTRACE_SYSCALL, pid, NULL, NULL);
        wait(&status);

        if ( WIFEXITED(status) ) {
            fprintf(stderr, "Target exited.\n");
            cont = 0;
            continue;
        }

        if (ptrace( PTRACE_GETREGS, pid, NULL, &regs) == -1 ) {
            fprintf(stderr, "Unable to read process registers\n");
```

```
            continue;
        }
        if ( state ) {
            /* system call trap */
            printf("System Call %X (%X, %X, %X, %X, %X)\n",
                    regs.orig_eax, regs.ebx, regs.ecx,
                    regs.edx, regs.esi, regs.edi );
        } else {
            printf("Return: %X\n", regs.orig_eax);
        }
      }
/*---------------------------------------------------------------------*/
```

Obviously, the output of this code would be tedious to use; a more sophisticated version would store a mapping of system call numbers (i.e., the index into the system call table of a particular entry) to their names, as well as a list of their parameters and return types.

The GNU BFD Library

GNU BFD is the GNU Binary File Descriptor library; it is shipped with the binutils package and is the basis for all of the included utilities, including objdump, objcopy, and ld. The reason sstripped binaries cannot be loaded by any of these utilities can be traced directly back to improper handling of the ELF headers by the BFD library. As a library for manipulating binaries, however, BFD is quite useful; it provides an abstraction of the object file, which allows file sections and symbols to be dealt with as distinct elements.

The BFD API could generously be described as unwieldy; hundreds of functions, inhumanly large structures, uncommented header files, and vague documentation—provided in the info format that the FSF still insists is a good idea—combine to drive away most programmers who might otherwise move on to write powerful binary manipulation tools.

To begin with, you must understand the BFD conception of a file. Every object file is in a specific format:

```
typedef enum bfd_format {
        bfd_unknown = 0,      /* file format is unknown */
        bfd_object,           /* linker/assember/compiler output */
        bfd_archive,          /* object archive file */
        bfd_core,             /* core dump */
        bfd_type_end          /* marks the end; don't use it! */
};
```

The format is determined when the file is opened for reading using `bfd_openr( )`. The format can be checked using the `bfd_check_format( )` routine. Once the file is loaded, details such as the specific file format, machine architecture, and endianness are all known and recorded in the `bfd` structure.

When a file is opened, the BFD library creates a bfd structure (defined in *bfd.h*), which is a bit large and has the following format:

```
struct bfd {
        const char                  *filename;
        const struct bfd_target     *xvec;
        void                        *iostream;
        boolean                      cacheable;
        boolean                      target_defaulted;
        struct _bfd                 *lru_prev, *lru_next;
        file_ptr                     where;
        boolean                      opened_once;
        boolean                      mtime_set;
        long                         mtime;
        int                          ifd;
        bfd_format                   format;
        enum bfd_direction           direction;
        flagword                     flags;
        file_ptr                     origin;
        boolean                      output_has_begun;
        struct sec                  *sections;
        unsigned int                 section_count;
        bfd_vma                      start_address;
        unsigned int                 symcount;
        struct symbol_cache_entry  **outsymbols;
        const struct bfd_arch_info  *arch_info;
        void                        *arelt_data;
        struct _bfd                 *my_archive;
        struct _bfd                 *next;
        struct _bfd                 *archive_head;
        boolean                      has_armap;
        struct _bfd                 *link_next;
        int                          archive_pass;
        union {
            struct aout_data_struct *aout_data;
            struct elf_obj_tdata    *elf_obj_data;
            /* ... */
        } tdata;
        void                        *usrdata;
        void                        *memory;
    };
```

This is the core definition of a BFD target; aside from the various management variables (xvec, iostream, cacheable, target_defaulted, etc.), the bfd structure contains the basic object file components, such as the entry point (start_address), sections, symbols, and relocations.

The first step when working with BFD is to be able to open and close a file reliably. This involves initializing BFD, calling an open function (one of the read-only functions bfd_openr, bfd_fdopenr, or bfd_openstreamr, or the write function bfd_openw), and closing the file with bfd_close:

```
/*-----------------------------------------------------------------------*/
    #include <errno.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/stat.h>
    #include <sys/types.h>

    #include <bfd.h>

    int main( int argc, char **argv ) {
              struct stat s;
        bfd *b;

          if ( argc < 2 ) {
                fprintf(stderr, "Usage: %s filename\n", argv[0]);
                return(1);
          }
          if ( stat( argv[1], &s) ) {
                fprintf(stderr, "Error: %s\n", strerror(errno) );
                return(2);
          }

        bfd_init( );
        b = bfd_openr( argv[1], NULL );
        if ( bfd_check_format(b, bfd_object ) ) {
            printf("Loading object file %s\n", argv[1]);
        } else if ( bfd_check_format(b, bfd_archive ) ) {
            printf("Loading archive file %s\n", argv[1]);
        }

        bfd_close(b);

          return(0);
    }
/*----------------------------------------------------------------------*/
```

How do you compile this monstrosity?

```
bash# gcc -I/usr/src/binutils/bfd -I/usr/src/binutils/include -o bfd \
>   -lbfd -liberty bfd.c
```

where */usr/src/binutils* is the location of the binutils source. While most distributions ship with a copy of binutils, the include files for those libraries are rarely present. If the standard include paths contain “dis-asm.h” and “bfd.h”, compilation will work fine without the binutils source code.

To the BFD library, an object file is just a linked list of sections, with file headers provided to enable traversing the list. Each section contains data in the form of code instructions, symbols, comments, dynamic linking information, or plain binary data. Detailed information about the object file, such as symbols and relocations, is associated with the `bfd` descriptor in order to make possible global modifications to sections.

The section structure is too large to be described here. It can be found among the 3,500 lines of *bfd.h*. The following routine demonstrates how to read the more interesting fields of the section structure for all sections in an object file.

```
/*-----------------------------------------------------------------------*/
    static void sec_print(bfd *b, asection *section, PTR jnk){
        unsigned char *buf;
        int i, j, size;

        printf( "%d %s\n",                 section->index, section->name );
        printf( "\tFlags 0x%08X",          section->flags );
        printf( "\tAlignment: 2^%d\n",     section->alignment_power );
        printf( "\tVirtual Address: 0x%X", section->vma );
        printf( "\tLoad Address: 0x%X\n",  section->lma );
        printf( "\tOutput Size: %4d",      section->_cooked_size );
        printf( "\tInput Size: %d\n",      section->_raw_size );

        size = section->_cooked_size;
        buf = calloc( size, 1 );
        if ( bfd_get_section_contents( b, section, buf, 0, size ) ) {
            printf("\n\tContents:\n");
            for ( i = 0; i < size; i +=16 ) {
                printf( "\t" );
                for (j = 0; j < 16 && j+i < size; j++ ) /* hex loop */
                    printf("%02X ", buf[i+j] );

                for ( ; j < 16; j++ )            /* pad loop */
                    printf("   ");

                for (j = 0; j < 16 && j+i < size; j++) /* ASCII loop */
                    printf("%c", isprint(buf[i+j])? buf[i+j] : '.');

                printf("\n");
            }
            printf("\n\n");
        }

        return;
    }

    int main( int argc, char **argv ) {
        /*     ... add this line before bfd_close()   */
        bfd_map_over_sections( b, sec_print, NULL );
        /*     ...     */
    }
/*---------------------------------------------------------------------*/
```

The only thing to notice here is the use of `bfd_map_over_sections( )`, which iterates over all sections in the file and invokes a callback for each section. Most of the section attributes can be accessed directly using the section structure or with BFD wrapper functions; the contents of a section, however, are not loaded until `bfd_get_section_contents( )` is called to explicitly copy the contents of a section (i.e., the code or data) to an allocated buffer.

Printing the contents of a file is fairly simple; however, BFD starts to earn its reputation when used to create output files. The process itself does not appear to be so difficult.

```
b1 = bfd_openr( input_file, NULL );
b2 = bfd_openw( output_file, NULL );
bfd_map_over_sections( b1, copy_section, b2 );
bfdclose( b2 );
bfdclose( b1 );
```

Seems simple, eh? Well, keep in mind this is GNU software.

To begin with, all sections in the output file must be defined before they can be filled with any data. This means two iterations through the sections already:

```
bfd_map_over_sections( b1, define_section, b2 );
bfd_map_over_sections( b1, copy_section, b2 );
```

In addition, the symbol table must be copied from one `bfd` descriptor to the other, and all of the relocations in each section must be moved over manually. This can get a bit clunky, as seen in the code below.

```
/*-----------------------------------------------------------------------*/
    #include <errno.h>
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/stat.h>
    #include <sys/types.h>
    #include <unistd.h>

    #include <bfd.h>

    /* return true for sections that will not be copied to the output file */
    static int skip_section( bfd *b, asection *s ) {
        /* skip debugging info */
        if ( (bfd_get_section_flags( b, s ) & SEC_DEBUGGING) )
            return( 1 );
        /* remove gcc cruft */
        if ( ! strcmp( s->name, ".comment" ) )
            return( 1 );
        if ( ! strcmp( s->name, ".note" ) )
            return( 1 );

        return(0);
    }

    struct COPYSECTION_DATA {
        bfd * output_bfd;
        asymbol **syms;
        int sz_syms, sym_count;
    };

    static void copy_section( bfd *infile, asection *section, PTR data ){
```

```
    asection *s;
    unsigned char *buf;
    long size, count, sz_reloc;
    struct COPYSECTION_DATA *d = data;
    bfd *outfile = d->output_bfd;
    asymbol **syms = d->syms;

    if ( skip_section( infile, section ) )
        return;

    /* get output section from input section struct */
    s = section->output_section;
    /* get sizes for copy */
    size = bfd_get_section_size_before_reloc (section );
    sz_reloc = bfd_get_reloc_upper_bound( infile, section );

    if ( ! sz_reloc ) {
        /* no relocations */
        bfd_set_reloc( outfile, s, (arelent **) NULL, 0);
    } else if ( sz_reloc  > 0 ) {
        /* build relocations */
        buf = calloc( sz_reloc, 1 );
        /* convert binary relocs to BFD internal representation */
        /* From info: "The SYMS table is also needed for horrible
           internal magic reasons". I kid you not.
           Welcome to hack city. */
        count = bfd_canonicalize_reloc(infile, section,
                              (arelent **)buf, syms );
        /* at this point, undesired symbols can be stripped */
        /* set the relocations for the output section */
        bfd_set_reloc( outfile, s, (arelent **) ((count) ?
                                 buf : NULL), count );
        free( buf );
    }

    /* here we manipulate BFD's private data for no apparent reason */
    section->_cooked_size = section->_raw_size;
    section->reloc_done = true;

    /* get input section contents, set output section contents */
    if ( section->flags & SEC_HAS_CONTENTS ) {
        buf = calloc( size, 1 );
        bfd_get_section_contents( infile, section, buf, 0, size );
        bfd_set_section_contents( outfile, s, buf, 0, size );
        free( buf );
    }
    return;
}


static void define_section( bfd *infile, asection *section, PTR data ){
    bfd *outfile = (bfd *) data;
    asection *s;
```

```
    if ( skip_section( infile, section ) )
        return;

    /* no idea why this is called "anyway"... */
    s = bfd_make_section_anyway( outfile, section->name );
    /* set size to same as infile section */
    bfd_set_section_size( outfile, s, bfd_section_size(infile,
                                   section) );
    /* set virtual address */
    s->vma =  section->vma;
    /* set load address */
    s->lma =  section->lma;
    /* set alignment -- the power 2 will be raised to */
    s->alignment_power = section->alignment_power;
    bfd_set_section_flags(outfile, s,
            bfd_get_section_flags(infile, section));
    /* link the output section to the input section -- don't ask why */
    section->output_section = s;
    section->output_offset = 0;
    /* copy any private BFD data from input to output section */
    bfd_copy_private_section_data( infile, section, outfile, s );
    return;
}

int file_copy( bfd *infile, bfd *outfile ) {
    struct COPYSECTION_DATA data = {0};

    if ( ! infile || ! outfile )    return(0);

    /* set output parameters to infile settings */
    bfd_set_format( outfile, bfd_get_format(infile) );
    bfd_set_arch_mach(outfile, bfd_get_arch(infile),
                    bfd_get_mach(infile));
    bfd_set_file_flags( outfile, bfd_get_file_flags(infile) &
                bfd_applicable_file_flags(outfile) );
    /* set the entry point of the output file */
    bfd_set_start_address( outfile, bfd_get_start_address(infile) );

    /* define sections for output file */
    bfd_map_over_sections( infile, define_section, outfile );

    /* get input file symbol table */
    data.sz_syms = bfd_get_symtab_upper_bound( infile );
    data.syms = calloc( data.sz_syms, 1 );

    /* convert binary symbol data to BFD internal format */
    data.sym_count = bfd_canonicalize_symtab( infile, data.syms );

    /* at this point the symbol table may be examined via
          for ( i=0; i < data.sym_count; i++ )
              asymbol *sym = data.syms[i];
       ...and so on, examining sym->name, sym->value, and sym->flags */

    /* generate output file symbol table */
```

```
    bfd_set_symtab( outfile, data.syms, data.sym_count );

    /* copy section content from input to output */
    data.output_bfd = outfile;
    bfd_map_over_sections( infile, copy_section, &data );

    /* copy whatever weird data BFD needs to make this a real file */
    bfd_copy_private_bfd_data( infile, outfile );
    return(1);
}

int main( int argc, char **argv ) {
        struct stat s;
    bfd *infile, *outfile;

      if ( argc < 3 ) {
              fprintf(stderr, "Usage: %s infile outfile\n", argv[0]);
            return(1);
      }
      if ( stat( argv[1], &s) ) {
            fprintf(stderr, "Error: %s\n", strerror(errno) );
            return(2);
      }

    bfd_init( );

    /* open input file for reading */
    infile = bfd_openr( argv[1], NULL );
    if ( ! infile ) {
        bfd_perror( "Error on infile" );
        return(3);
    }
    /* open output file for writing */
    outfile = bfd_openw( argv[2], NULL );
    if ( ! outfile ) {
        bfd_perror( "Error on outfile" );
        return(4);
    }

    if ( bfd_check_format (infile, bfd_object ) ) {
        /* routine that does all the work */
        file_copy( infile, outfile );
    } else if ( bfd_check_format(infile, bfd_archive ) ) {
        fprintf( stderr, "Error: archive files not supported\n");
        return(5);
    }

    bfd_close(outfile);
    bfd_close(infile);

      return(0);
}
/*-----------------------------------------------------------------------*/
```

This utility will strip the `.comment` and `.note` sections from an ELF executable:

```
bash# gcc -I/usr/src/binutils/bfd -I/usr/src/binutils/include \
>    -o bfdtest -lbfd -liberty bfd.c
bash# ./bfdtest a.out a.out.2
bash# objdump -h a.out | grep .comment
23 .comment      00000178  00000000  00000000  00001ff0  2**0
bash# objdump -h tst | grep .comment
bash#
```

With some work, this could be improved to provide an advanced ELF stripper (now there's a name that leaps out of the manpage) such as sstrip(1), or it could be rewritten to add code into an existing ELF executable in the manner of objcopy and ld.

Disassembling with libopcodes

The libopcodes library, like much of the GNU code intended only for internal use, requires hackish and inelegant means (e.g., global variables, replacement fprintf(3) routines) to get it working. The result is ugly to look at and may get a bit dodgy when threaded—but it's free, and it's a disassembler.

In a nutshell, one uses libopcodes by including the file dis-asm.h from the binutils distribution, filling a disassemble_info structure, and calling either `print_insn_i386_att()` or `print_insn_i386_intel()`.

The `disassemble_info` structure is pretty large and somewhat haphazard in design; it has the following definition (cleaned up from the actual header):

```
typedef int (*fprintf_ftype) (FILE *, const char*, ...);
typedef int (*read_memory_func_t) (bfd_vma memaddr, bfd_byte *myaddr,
            unsigned int length, struct disassemble_info *info);
typedef void (*memory_error_func_t) (int status, bfd_vma memaddr,
            struct disassemble_info *info);
typedef void (*print_address_func_t) (bfd_vma addr,
            struct disassemble_info *info);
typedef int (*symbol_at_address_func_t) (bfd_vma addr,
            struct disassemble_info * info);

typedef struct disassemble_info {
      fprintf_ftype             fprintf_func;
      unsigned char            *stream;
      void                     *application_data;
      enum bfd_flavour          flavour;
      enum bfd_architecture     arch;
      unsigned long             mach;
      enum bfd_endian           endian;
      asection                 *section;
      asymbol                 **symbols;
      int                       num_symbols;
      unsigned long             flags;
      void                     *private_data;
      read_memory_func_t        read_memory_func;
```

```
    memory_error_func_t          memory_error_func;
    print_address_func_t         print_address_func;
    symbol_at_address_func_t     symbol_at_address_func;
    bfd_byte                    *buffer;
    bfd_vma                      buffer_vma;
    unsigned int                 buffer_length;
    int                          bytes_per_line;
    int                          bytes_per_chunk;
    enum bfd_endian              display_endian;
    unsigned int                 octets_per_byte;
    char                         insn_info_valid;
    char                         branch_delay_insns;
    char                         data_size;
    enum dis_insn_type           insn_type;
    bfd_vma                      target;
    bfd_vma                      target2;
    char *                       disassembler_options;
} disassemble_info;
```

Some of these fields (e.g., `flavour`, `section`, `symbols`) duplicate the data managed by the BFD library and are in fact unused by the disassembler, some are internal to the disassembler (e.g., `private_data`, `flags`), some are the necessarily pedantic information required to support disassembly of binary files from another platform (e.g., `arch`, `mach`, `endian`, `display_endian`, `octets_per_byte`), and some are actually not used at all in the x86 disassembler (e.g., `insn_info_valid`, `branch_delay_insns`, `data_size`, `insn_type`, `target`, `target2`, `disassembler_options`).

The enumerations are defined in *bfd.h*, supplied with binutils; note that `flavour` refers to the file format and can get set to `unknown`. The `endian` and `arch` fields should be set to their correct values. The definitions are as follows:

```
enum bfd_flavour {
    bfd_target_unknown_flavour, bfd_target_aout_flavour,
    bfd_target_coff_flavour, bfd_target_ecoff_flavour,
    bfd_target_xcoff_flavour, bfd_target_elf_flavour,bfd_target_ieee_flavour,
    bfd_target_nlm_flavour, bfd_target_oasys_flavour,
    bfd_target_tekhex_flavour, bfd_target_srec_flavour,
    bfd_target_ihex_flavour, bfd_target_som_flavour, bfd_target_os9k_flavour,
    bfd_target_versados_flavour, bfd_target_msdos_flavour,
    bfd_target_ovax_flavour, bfd_target_evax_flavour
    };

    enum bfd_endian { BFD_ENDIAN_BIG,BFD_ENDIAN_LITTLE, BFD_ENDIAN_UNKNOWN};

    enum bfd_architecture {
      bfd_arch_unknown, bfd_arch_obscure, bfd_arch_m68k, bfd_arch_vax,
      bfd_arch_i960, bfd_arch_a29k, bfd_arch_sparc, bfd_arch_mips,
      bfd_arch_i386, bfd_arch_we32k, bfd_arch_tahoe, bfd_arch_i860,
      bfd_arch_i370, bfd_arch_romp, bfd_arch_alliant, bfd_arch_convex,
      bfd_arch_m88k, bfd_arch_pyramid, bfd_arch_h8300, bfd_arch_powerpc,
      bfd_arch_rs6000, bfd_arch_hppa, bfd_arch_d10v, bfd_arch_d30v,
      bfd_arch_m68hc11, bfd_arch_m68hc12, bfd_arch_z8k, bfd_arch_h8500,
```

```
    bfd_arch_sh, bfd_arch_alpha, bfd_arch_arm, bfd_arch_ns32k, bfd_arch_w65,
    bfd_arch_tic30, bfd_arch_tic54x, bfd_arch_tic80, bfd_arch_v850,
    bfd_arch_arc, bfd_arch_m32r, bfd_arch_mn10200, bfd_arch_mn10300,
    bfd_arch_fr30, bfd_arch_mcore, bfd_arch_ia64,  bfd_arch_pj, bfd_arch_avr,
    bfd_arch_cris, bfd_arch_last
  };
```

The mach field is an extension to the arch field; constants are defined (in the definition of the bfd_architecture enum in *bfd.h*) for various CPU architectures. The Intel ones are:

```
#define bfd_mach_i386_i386 0
#define bfd_mach_i386_i8086 1
#define bfd_mach_i386_i386_intel_syntax 2
#define bfd_mach_x86_64 3
#define bfd_mach_x86_64_intel_syntax 4
```

This is more than a little strange, since Intel IA64 has its own arch type. Note that setting the mach field to bfd_mach_i386_i386_intel_syntax has no effect on the output format; you must call the appropriate print_insn routine, which sets the output format strings to AT&T or Intel syntax before calling print_insn_i386( ).

The disassemble_info structure should be initialized to zero, then manipulated either directly or with one of the provided macros:

```
#define INIT_DISASSEMBLE_INFO(INFO, STREAM, FPRINTF_FUNC)
```

where INFO is the *static* address of the struct (i.e., not a pointer—the macro uses "INFO." to access struct fields, not "INFO->"), STREAM is the file pointer passed to fprintf(), and FPRINTF_FUNC is either fprintf( ) or a replacement with the same syntax.

Why is fprintf( ) needed? It is assumed by libopcodes that the disassembly is going to be immediately printed with no intervening storage or analysis. This means that to store the disassembly for further processing, you must replace fprintf( ) with a custom function that builds a data structure for the instruction.

This is not as simple as it sounds, however. The fprintf( ) function is called once for the mnemonic and once for each operand in the instruction; as a result, any fprintf( ) replacement is going to be messy:

```
char mnemonic[32] = {0}, src[32] = {0}, dest[32] = {0}, arg[32] = {0};

    int disprintf(FILE *stream, const char *format, ...){
        va_list args;
        char *str;

        va_start (args, format);
        str = va_arg(args, char*);

        if ( ! mnemonic[0] ) {
            strncpy(mnemonic, str, 31);
        } else if ( ! src[0] ) {
```

```
            strncpy(src, str, 31);
        } else if ( ! dest[0] ) {
            strncpy(dest, str, 31);
        } else {
            if ( ! strcmp( src, dest ) )
                strncpy(dest, str, 31);
            else
                strncpy(arg, str, 31);
        }
        va_end (args);

        return(0);
    }
```

Simple, graceful, elegant, right? No. The `src` argument occasionally gets passed twice, requiring the `strcmp( )` in the else block. Note that the string buffers must be zeroed out after every successful disassembly in order for `disprintf( )` to work at all.

Despite the size of the `disassemble_info` structure, not much needs to be set in order to use libopcodes. The following code properly initializes the structure:

```
/* target settings */
    info.arch             = bfd_arch_i386;
    info.mach             = bfd_mach_i386_i386;
    info.flavour          = bfd_target_unknown_flavour;
    info.endian           = BFD_ENDIAN_LITTLE;
    /* display/output settings */
    info.display_endian   = BFD_ENDIAN_LITTLE;
    info.fprintf_func     = fprintf;
    info.stream           = stdout;
    /* what to disassemble */
    info.buffer           = buf;        /* buffer of bytes to disasm */
    info.buffer_length    = buf_len;    /* size of buffer */
    info.buffer_vma       = buf_vma;    /* base RVA of buffer */
```

The disassembler can now enter a loop, calling the appropriate `print_insn` routine until the end of the buffer to be disassembled is reached:

```
unsigned int pos = 0;
while ( pos < info.buffer_length ) {
    printf("%8X : ", info.buffer_vma + pos);
    pos += print_insn_i386_intel(info.buffer_vma + pos, &info);
    printf("\n");
}
```

The following program implements a libopcodes-based disassembler, using BFD to load the file and providing a replacement `fprintf( )` routine based on the above `disprintf( )` routine. The code can be compiled with:

```
gcc -I/usr/src/binutils/bfd -I/usr/src/binutils/include -o bfd \
-lbfd -liberty -lopcodes bfd.c
```

Note that it requires the BFD libraries as well as libopcodes; this is largely in order to tie the code in with the discussion of BFD in the previous section, as libopcodes can be used without BFD simply by filling the disassemble_info structure with NULL values instead of BFD type information.

```
/*---------------------------------------------------------------------------*/
    #include <errno.h>
    #include <fcntl.h>
    #include <stdarg.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/stat.h>
    #include <sys/types.h>
    #include <unistd.h>

    #include <bfd.h>
    #include <dis-asm.h>


    struct ASM_INSN {
        char mnemonic[16];
        char src[32];
        char dest[32];
        char arg[32];
    } curr_insn;

    int disprintf(FILE *stream, const char *format, ...){
        /* Replacement fprintf( ) for libopcodes.
         * NOTE: the following assumes src, dest order from disassembler */
        va_list args;
        char *str;

        va_start (args, format);
        str = va_arg(args, char*);

        /* this sucks, libopcodes passes one mnem/operand per call --
         * and passes src twice */
        if ( ! curr_insn.mnemonic[0] ) {
            strncpy(curr_insn.mnemonic, str, 15);
        } else if ( ! curr_insn.src[0] ) {
            strncpy(curr_insn.src, str, 31);
        } else if ( ! curr_insn.dest[0] ) {
            strncpy(curr_insn.dest, str, 31);
            if (strncmp(curr_insn.dest, "DN", 2) == 0)
                    curr_insn.dest[0] = 0;
        } else {
            if ( ! strcmp( curr_insn.src, curr_insn.dest ) ) {
                /* src was passed twice */
                strncpy(curr_insn.dest, str, 31);
            } else {
                strncpy(curr_insn.arg, str, 31);
            }
        }
        va_end (args);

        return(0);
    }
```

```
void print_insn( void ) {
    printf("\t%s", curr_insn.mnemonic);
    if ( curr_insn.src[0] ) {
        printf("\t%s", curr_insn.src );
        if ( curr_insn.dest[0] ) {
            printf(", %s", curr_insn.dest );
            if ( curr_insn.arg[0] ) {
                printf(", %s", curr_insn.arg );
            }
        }
    }
    return;
}

int disassemble_forward( disassembler_ftype disassemble_fn,
                disassemble_info *info, unsigned long rva ) {
    int bytes = 0;

    while ( bytes < info->buffer_length ) {
        /* call the libopcodes disassembler */
        memset( &curr_insn, 0, sizeof( struct ASM_INSN ));
        bytes += (*disassemble_fn)(info->buffer_vma + bytes, info);

        /* -- print any symbol names as labels here -- */
        /* print address of instruction */
        printf("%8X : ", info->buffer_vma + bytes);
        /* -- analyze disassembled instruction here -- */
        print_insn( );
        printf("\n");
    }
    return( bytes );
}

int disassemble_buffer( disassembler_ftype disassemble_fn,
                disassemble_info *info ) {
    int i, size, bytes = 0;

    while ( bytes < info->buffer_length ) {
        /* call the libopcodes disassembler */
        memset( &curr_insn, 0, sizeof( struct ASM_INSN ));
        size = (*disassemble_fn)(info->buffer_vma + bytes, info);

        /* -- analyze disassembled instruction here -- */

        /* -- print any symbol names as labels here -- */
        printf("%8X:   ", info->buffer_vma + bytes);
        /* print hex bytes */
        for ( i = 0; i < 8; i++ ) {
            if ( i < size )
                printf("%02X ", info->buffer[bytes + i]);
            else
                printf("   ");
        }
        print_insn( );
```

```
        printf("\n");
        bytes += size;    /* advance position in buffer */
    }
    return( bytes );
}

static void disassemble( bfd *b, asection *s, unsigned char *buf,
                    int size, unsigned long buf_vma ) {
    disassembler_ftype disassemble_fn;
    static disassemble_info info = {0};

    if ( ! buf )     return;
    if ( ! info.arch ) {
        /* initialize everything */
        INIT_DISASSEMBLE_INFO(info, stdout, disprintf);
        info.arch = bfd_get_arch(b);
        info.mach = bfd_mach_i386_i386;    /* BFD_guess no worka */
        info.flavour = bfd_get_flavour(b);
        info.endian = b->xvec->byteorder;
        /* these can be replaced with custom routines
          info.read_memory_func = buffer_read_memory;
          info.memory_error_func = perror_memory;
          info.print_address_func = generic_print_address;
          info.symbol_at_address_func = generic_symbol_at_address;
          info.fprintf_func = disprintf; //handled in macro above
          info.stream = stdout;          // ditto
          info.symbols = NULL;
          info.num_symbols = 0;
        */
        info.display_endian = BFD_ENDIAN_LITTLE;

    }

    /* choose disassembler function */
    disassemble_fn = print_insn_i386_att;
    /* disassemble_fn = print_insn_i386_intel; */
    /* these are section dependent */
    info.section = s;            /* section to disassemble */
    info.buffer = buf;        /* buffer of bytes to disassemble */
    info.buffer_length = size;    /* size of buffer */
    info.buffer_vma = buf_vma;    /* base RVA of buffer */

    disassemble_buffer( disassemble_fn, &info );

    return;
}

static void print_section_header( asection *s, const char *mode ) {

    printf("Disassembly of section %s as %s\n", s->name, mode );
    printf("RVA: %08X LMA: %08X Flags: %08X Size: %X\n", s->vma,
            s->lma, s->flags, s->_cooked_size );
    printf( "---------------------------------------------------------"
          "----------------------\n" );
```

```
    return;
}

static void disasm_section_code( bfd *b, asection *section ) {
    int size;
    unsigned char *buf;

    size = bfd_section_size( b, section );
    buf = calloc( size, 1 );
    if (! buf || ! bfd_get_section_contents(b, section, buf, 0, size ))
        return;
    print_section_header( section, "code" );
    disassemble( b, section, buf, size, section->vma );
    printf("\n\n");
    free( buf );
    return;
}

static void disasm_section_data( bfd *b, asection *section ) {
    int i, j, size;
    unsigned char *buf;

    size = bfd_section_size( b, section );
    buf = calloc( size, 1 );
    if ( ! bfd_get_section_contents( b, section, buf, 0, size ) )
        return;
    print_section_header( section, "data" );
    /* do hex dump of data */
    for ( i = 0; i < size; i +=16 ) {
        printf( "%08X:   ", section->vma + i );
        for (j = 0; j < 16 && j+i < size; j++ )
            printf("%02X ", buf[i+j] );
        for ( ; j < 16; j++ )
            printf("   ");
        printf("  ");
        for (j = 0; j < 16 && j+i < size; j++ )
            printf("%c", isprint(buf[i+j]) ? buf[i+j] : '.' );
        printf("\n");
    }
    printf("\n\n");
    free( buf );
    return;
}

static void disasm_section( bfd *b, asection *section, PTR data ){
    if ( ! section->flags & SEC_ALLOC )  return;
    if ( ! section->flags & SEC_LOAD ) return;
    if ( section->flags & SEC_LINKER_CREATED ) return;
    if ( section->flags & SEC_CODE) {
        if ( ! strncmp(".plt", section->name, 4) ||
             ! strncmp(".got", section->name, 4) ) {
            return;
        }
        disasm_section_code( b, section );
```

```
    } else if ( (section->flags & SEC_DATA ||
             section->flags & SEC_READONLY) &&
            section->flags & SEC_HAS_CONTENTS  ) {
        disasm_section_data( b, section );
    }
    return;
}

int main( int argc, char **argv ) {
          struct stat s;
    bfd *infile;

      if ( argc < 2 ) {
            fprintf(stderr, "Usage: %s target\n", argv[0]);
            return(1);
      }
      if ( stat( argv[1], &s) ) {
            fprintf(stderr, "Error: %s\n", strerror(errno) );
            return(2);
      }

    bfd_init( );

    /* open input file for reading */
    infile = bfd_openr( argv[1], NULL );
    if ( ! infile ) {
        bfd_perror( "Error on infile" );
        return(3);
    }

    if ( bfd_check_format (infile, bfd_object ) ||
          bfd_check_format(infile, bfd_archive ) ) {
        bfd_map_over_sections( infile, disasm_section, NULL );
    } else  {
        fprintf( stderr, "Error: unknown file format\n");
        return(5);
    }

    bfd_close(infile);

      return(0);
}
/*-----------------------------------------------------------------------*/
```

As disassemblers go, this is rather mundane—and it's not an improvement on objdump. Being BFD-based, it does not perform proper loading of the ELF file header and is therefore still unable to handle sstriped binaries—however, this could be fixed by removing the dependence on BFD and using a custom ELF file loader.

The disassembler could also be improved by adding the ability to disassemble based on the flow of execution, rather than on the sequence of addresses in the code section. The next program combines libopcodes with the instruction types presented earlier in "Intermediate Code Generation." The result is a disassembler that records operand type information and uses the mnemonic to determine if the instruction influences the flow of execution, and thus whether it should follow the target of the instruction.

```
/*---------------------------------------------------------------------------*/
    #include <errno.h>
    #include <fcntl.h>
    #include <stdarg.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/stat.h>
    #include <sys/types.h>
    #include <unistd.h>

    #include <bfd.h>
    #include <dis-asm.h>


    /* operand types */
    enum op_type { op_unk, op_reg, op_imm, op_expr, op_bptr, op_dptr,
               op_wptr };

    struct ASM_INSN {
        char mnemonic[16];
        char src[32];
        char dest[32];
        char arg[32];
        enum op_type src_type, dest_type, arg_type;
    } curr_insn;


    enum op_type optype( char *op){
        if ( op[0] == '%' ) { return(op_reg); }
        if ( op[0] == '$' ) { return(op_imm); }
        if ( strchr(op, '(')) { return(op_expr); }
        if ( strncmp( op, "BYTE PTR", 8)) { return(op_bptr); }
        if ( strncmp( op, "DWORD PTR", 9)) { return(op_dptr); }
        if ( strncmp( op, "WORD PTR", 8)) { return(op_wptr); }
        return(op_unk);
    }

    /* we kind of cheat with these, since Intel has so few 'j' insns */
    #define JCC_INSN 'j'
    #define JMP_INSN "jmp"
    #define LJMP_INSN "ljmp"
    #define CALL_INSN "call"
    #define RET_INSN "ret"

    enum flow_type { flow_branch, flow_cond_branch, flow_call, flow_ret,
                 flow_none };

    enum flow_type insn_flow_type( char *insn ) {
        if (! strncmp( JMP_INSN, insn, 3)  ||
            ! strncmp( LJMP_INSN, insn, 4) ) {
            return( flow_branch );
```

```
    } else if ( insn[0] == JCC_INSN ) {
        return( flow_cond_branch ) ;
    } else if ( ! strncmp( CALL_INSN, insn, 4) ) {
        return( flow_call );
    } else if ( ! strncmp( RET_INSN, insn, 3) ) {
        return( flow_ret );
    }
    return( flow_none );
}

int disprintf(FILE *stream, const char *format, ...){
    va_list args;
    char *str;

    va_start (args, format);
    str = va_arg(args, char*);

    /* this sucks, libopcodes passes one mnem/operand per call --
     * and passes src twice */
    if ( ! curr_insn.mnemonic[0] ) {
        strncpy(curr_insn.mnemonic, str, 15);
    } else if ( ! curr_insn.src[0] ) {
        strncpy(curr_insn.src, str, 31);
        curr_insn.src_type = optype(curr_insn.src);
    } else if ( ! curr_insn.dest[0] ) {
        strncpy(curr_insn.dest, str, 31);
        curr_insn.dest_type = optype(curr_insn.dest);
        if (strncmp(curr_insn.dest, "DN", 2) == 0)
                curr_insn.dest[0] = 0;
    } else {
        if ( ! strcmp( curr_insn.src, curr_insn.dest ) ) {
            /* src was passed twice */
            strncpy(curr_insn.dest, str, 31);
            curr_insn.dest_type = optype(curr_insn.dest);
        } else {
            strncpy(curr_insn.arg, str, 31);
            curr_insn.arg_type = optype(curr_insn.arg);
        }
    }
    va_end (args);

    return(0);
}

void print_insn( void ) {
    printf("\t%s", curr_insn.mnemonic);
    if ( curr_insn.src[0] ) {
        printf("\t%s", curr_insn.src );
        if ( curr_insn.dest[0] ) {
            printf(", %s", curr_insn.dest );
            if ( curr_insn.arg[0] ) {
                printf(", %s", curr_insn.arg );
            }
        }
    }
```

```
    return;
}

int rva_from_op( char *op, unsigned long *rva ) {
    if ( *op == '*' )     return(0);    /* pointer */
    if ( *op == '$' )     op++;
    if ( isxdigit(*op) ) {
        *rva = strtoul(curr_insn.src, NULL, 16);
        return(1);
    }
    return(0);
}

static void disassemble( bfd *b, unsigned long rva, int nest );

int disassemble_forward( disassembler_ftype disassemble_fn,
            disassemble_info *info, unsigned long rva, int nest ) {
    int i, good_rva, size, offset, bytes = 0;
    unsigned long branch_rva;
    enum flow_type flow;

    if (! nest )
        return(0);

    /* get offset of rva into section */
    offset = rva - info->buffer_vma;

    /* prevent runaway loops */
    nest--;

    while ( bytes < info->buffer_length ) {
        /* this has to be verified because of branch following */
        if ( rva < info->buffer_vma ||
                rva >= info->buffer_vma + info->buffer_length ) {
            /* recurse via disassemble( ) then exit */
            disassemble( NULL, rva + bytes, nest );
            return(0);
        }

        /* call the libopcodes disassembler */
        memset( &curr_insn, 0, sizeof( struct ASM_INSN ));
        size = (*disassemble_fn)(rva + bytes, info);

        /* -- analyze disassembled instruction here -- */

        /* -- print any symbol names as labels here -- */
        printf("%8X:   ", rva + bytes);
        /* print hex bytes */
        for ( i = 0; i < 8; i++ ) {
            if ( i < size )
                printf("%02X ", info->buffer[offset+bytes+i]);
            else
                printf("   ");
        }
```

```
        print_insn( );
        printf("\n");
        bytes += size;    /* advance position in buffer */
        /* check insn type */
        flow = insn_flow_type( curr_insn.mnemonic );
        if ( flow == flow_branch || flow == flow_cond_branch ||
             flow == flow_call ) {
            /* follow branch branch */
            good_rva = 0;
            if ( curr_insn.src_type == op_bptr ||
                 curr_insn.src_type == op_wptr ||
                 curr_insn.src_type == op_dptr    ) {
                good_rva = rva_from_op( curr_insn.src,
                            &branch_rva );
            }
            if ( good_rva ) {
                printf(";------------------ FOLLOW BRANCH %X\n",
                        branch_rva );
                disassemble_forward( disassemble_fn, info,
                            branch_rva, nest );
            }
        }
        if ( flow == flow_branch || flow == flow_ret ) {
            /* end of execution flow : exit loop */
            bytes = info->buffer_length;
            printf(";------------------------- END BRANCH\n");
            continue;
        }
    }
    return( bytes );
}

struct RVA_SEC_INFO {
    unsigned long rva;
    asection *section;
};

static void find_rva( bfd *b, asection *section, PTR data ){
    struct RVA_SEC_INFO *rva_info = data;
    if ( rva_info->rva >= section->vma &&
         rva_info->rva < section->vma + bfd_section_size( b, section ))
        /* we have a winner */
        rva_info->section = section;
    return;
}

static void disassemble( bfd *b, unsigned long rva, int nest ) {
    static disassembler_ftype disassemble_fn;
    static disassemble_info info = {0};
    static bfd *bfd = NULL;
    struct RVA_SEC_INFO rva_info;
    unsigned char *buf;
    int size;
```

```
    if ( ! bfd ) {
        if ( ! b )     return;
        bfd = b;
        /* initialize everything */
        INIT_DISASSEMBLE_INFO(info, stdout, disprintf);
        info.arch = bfd_get_arch(b);
        info.mach = bfd_mach_i386_i386;
        info.flavour = bfd_get_flavour(b);
        info.endian = b->xvec->byteorder;
        info.display_endian = BFD_ENDIAN_LITTLE;
        disassemble_fn = print_insn_i386_att;
    }

    /* find section containing rva */
    rva_info.rva = rva;
    rva_info.section = NULL;
    bfd_map_over_sections( bfd, find_rva, &rva_info );
    if ( ! rva_info.section )
        return;
    size = bfd_section_size( bfd, rva_info.section );
    buf = calloc( size, 1 );
    /* we're gonna be mean here and only free the calloc at exit() */
    if ( ! bfd_get_section_contents( bfd, rva_info.section, buf, 0,
                                    size ) )
        return;

    info.section = rva_info.section;           /* section to disasm */
    info.buffer = buf;                         /* buffer to disasm */
    info.buffer_length = size;                 /* size of buffer */
    info.buffer_vma = rva_info.section->vma;   /* base RVA of buffer */

    disassemble_forward( disassemble_fn, &info, rva, nest );

    return;
}

static void print_section_header( asection *s, const char *mode ) {

    printf("Disassembly of section %s as %s\n", s->name, mode );
    printf("RVA: %08X LMA: %08X Flags: %08X Size: %X\n", s->vma,
           s->lma, s->flags, s->_cooked_size );
    printf( "---------------------------------------------------------"
          "------------------------\n" );
    return;
}

static void disasm_section( bfd *b, asection *section, PTR data ){
    int i, j, size;
    unsigned char *buf;

    /* we only care about data sections */
    if ( ! section->flags & SEC_ALLOC )  return;
    if ( ! section->flags & SEC_LOAD ) return;
    if ( section->flags & SEC_LINKER_CREATED ) return;
```

```
    if ( section->flags & SEC_CODE) {
        return;
    } else if ( (section->flags & SEC_DATA ||
              section->flags & SEC_READONLY) &&
             section->flags & SEC_HAS_CONTENTS  ) {
        /* print dump of data section */
        size = bfd_section_size( b, section );
        buf = calloc( size, 1 );
        if ( ! bfd_get_section_contents( b, section, buf, 0, size ) )
            return;
        print_section_header( section, "data" );
        for ( i = 0; i < size; i +=16 ) {
            printf( "%08X:   ", section->vma + i );
            for (j = 0; j < 16 && j+i < size; j++ )
                printf("%02X ", buf[i+j] );
            for ( ; j < 16; j++ )
                printf("   ");
            printf("  ");
            for (j = 0; j < 16 && j+i < size; j++ )
                printf("%c", isprint(buf[i+j]) ? buf[i+j] : '.');
            printf("\n");
        }
        printf("\n\n");
        free( buf );
    }
    return;
}

int main( int argc, char **argv ) {
      struct stat s;
    bfd *infile;

      if ( argc < 2 ) {
            fprintf(stderr, "Usage: %s target\n", argv[0]);
            return(1);
      }
          if ( stat( argv[1], &s) ) {
                fprintf(stderr, "Error: %s\n", strerror(errno) );
                return(2);
          }

    bfd_init( );

    /* open input file for reading */
    infile = bfd_openr( argv[1], NULL );
    if ( ! infile ) {
        bfd_perror( "Error on infile" );
        return(3);
    }

    if ( bfd_check_format (infile, bfd_object ) ||
          bfd_check_format(infile, bfd_archive ) ) {
        /* disassemble forward from entry point */
        disassemble( infile, bfd_get_start_address(infile), 10 );
```

```
            /* disassemble data sections */
            bfd_map_over_sections( infile, disasm_section, NULL );
        } else  {
            fprintf( stderr, "Error: unknown file format\n");
            return(5);
        }

        bfd_close(infile);

          return(0);
    }
/*-----------------------------------------------------------------------*/
```

Granted, this has its problems. The fprintf-based nature of the output means that instructions are printed as they are disassembled, rather than in address order; a better implementation would be to add each disassembled instruction to a linked list or tree, then print once all disassembly and subsequent analysis has been performed. Furthermore, since previously disassembled addresses are not stored, the only way to prevent endless loops is by using an arbitrary value to limit recursion. The disassembler relies on a single shared `disassemble_info` structure rather than providing its own, making for some messy code where the branch following causes a recursion into a different section (thereby overwriting `info->buffer`, `info->buffer_vma`, `info->buffer->size`, and `info->section`). Not an award-winning design to be sure; it cannot even follow function pointers!

As an example, however, it builds on the code of the previous disassembler to demonstrate how to implement branch following during disassembly. At this point, the program is no longer a trivial objdump-style disassembler; further development would require some intermediate storage of the disassembled instructions, as well as more intelligent instruction analysis. The instruction types can be expanded and used to track cross-references, monitor stack position, and perform algorithm analysis. A primitive virtual machine can be implemented by simulating reads and writes to addresses and registers, as indicated by the operand type.

Modifications such as these are beyond the scope of a simple introduction and are not illustrated here; hopefully, the interested reader has found enough information here to pursue such projects with confidence.

References

- Linux on the Half-ELF." *Mammon_'s Tales to his Grandson*:
- Packet Storm: Linux reverse-engineering tools. (*http://packetstormsecurity.org/linux/reverse-engineering/*)
- Sourceforge: open source development projects. (*http://www.sourceforge.net*)
- Freshmeat: Linux and open source software. (*http://www.freshmeat.net*)
- Debugging with GDB. (*http://www.gnu.org/manual/gdb-5.1.1/html_chapter/gdb_toc.html*)

- GDB Quick Reference Card. (*http://www.refcards.com/about/gdb.html*)
- Linux Assembly. (*http://linuxassembly.org*)
- Silvio Cesare: Coding. (*http://www.big.net.au/~silvio/coding/*)
- Hooking Interrupt and Exception Handlers in Linux. (*http://www.eccentrix.com/members/mammon/Text/linux_hooker.txt*)
- Muppet Labs: ELF Kickers. (*http://www.muppetlabs.com/~breadbox/software/elfkickers.html)*
- Tools and Interface Standards: The Executable Linkable Format. (*http://developer.intel.com/vtune/tis.htm*)
- LIB BFD, the Binary File Descriptor Library. (*http://www.gnu.org/manual/bfd-2.9.1/html_chapter/bfd_toc.html*)
- *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set. (http://www.intel.com/design/pentiumii/manuals/* and *http://www.intel.com/design/litcentr/index.htm*)