

## Reverse Engineering a Program

In this last section, we will be writing a simple hello world program in C, compiling it, then analyzing the disassembled binary. The code will be compiled with gcc and disassembled using gdb; if you are using Windows, you can get Dev-C++ from [bloodshed.net](http://bloodshed.net) which is a nice IDE that comes with all the gcc utilities, including gdb. Bear in mind that if you compile the source code yourself, your assembly code may be slightly different from mine due to variations in the different versions of gcc (I am using gcc v3.3.5 on Linux and v3.4.2 on Windows – they both produce identical assembly instructions). Also, your memory addresses probably won't match mine, but this is normal as they will usually be different when compiled on different systems. Finally, we will examine the disassembly of a slightly more complex program and walk through reverse engineering it.

### Using GDB

As stated earlier, gdb is both a debugger and a disassembler. In the following examples, we will be using gdb as a disassembler to perform a static analysis of our code. Gdb has many commands, but for our purposes there are just a few we will be using:

Command	Example	Explanation
file	file helloworld	Open the specified program in gdb. The program name can also be specified on the command line when starting gdb (\$gdb helloworld).
disassemble	disassemble main	Disassemble the specified function in the program. Gdb will display the function's assembly instructions on screen.
x	x/20s 0x80403001	Examine the contents of 20 addresses as strings starting at memory address 0x80403001. If you want to view the contents in hexadecimal, replace the 's' with an 'x'.

### Hello World

We will first use gdb to analyze a binary compiled from the following source code:

```
int main(int argc, char *argv[])
{
    printf("Hello World!\n");
    return 0;
}
```

Save this program as helloworld.c and compile it with 'gcc -o helloworld helloworld.c'; run the resulting binary and it should print "Hello World!" on the screen and exit. So far so good, now let's take a look at the assembly code:

heff@TPad:~/Programming\$ ***gdb helloworld***

*GNU gdb 6.3-debian*

*Copyright 2004 Free Software Foundation, Inc.*

*GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.*

*Type "show copying" to see the conditions.*

*There is absolutely no warranty for GDB. Type "show warranty" for details.*

*This GDB was configured as "i386-linux"...Using host libthread\_db library "/lib/libthread\_db.so.1".*

***(gdb) disassemble main***

*Dump of assembler code for function main:*

```
0x08048384 <main+0>:  push  %ebp
0x08048385 <main+1>:  mov   %esp,%ebp
0x08048387 <main+3>:  sub   $0x8,%esp
0x0804838a <main+6>:  and   $0xffffffff0,%esp
0x0804838d <main+9>:  mov   $0x0,%eax
0x08048392 <main+14>: sub   %eax,%esp
0x08048394 <main+16>: movl  $0x80484c4,(%esp)
0x0804839b <main+23>: call  0x80482b0 <_init+56>
0x080483a0 <main+28>: mov   $0x0,%eax
0x080483a5 <main+33>: leave
0x080483a6 <main+34>: ret
```

*End of assembler dump.*

Let's look at each instruction, keeping in mind that this disassembly is in the AT&T syntax (source on the left, destination on the right):

```
0x08048384 <main+0>:  push  %ebp
0x08048385 <main+1>:  mov   %esp,%ebp
0x08048387 <main+3>:  sub   $0x8,%esp
```

These three instructions should be familiar; they are the function's prologue. The '*push %ebp*' instruction saves the current EBP value onto the stack; '*mov %esp,%ebp*' creates the new EBP value by copying ESP into EBP; then eight bytes of space is created on the stack for local variables using the '*sub \$0x8,%esp*' instruction.

```
0x0804838a <main+6>:  and   $0xffffffff0,%esp
0x0804838d <main+9>:  mov   $0x0,%eax
0x08048392 <main+14>: sub   %eax,%esp
```

These three instructions are used to clean up any stray bits and prepare ESP and the stack at the beginning of the program; they are present only in a program's *main()* function, but not any subsequent functions. The first command zeros out the last byte of the value in ESP; the next two commands put the value 0 into the EAX register, then subtracts the EAX register (aka, zero) from the stack pointer.

```
0x08048394 <main+16>: movl  $0x80484c4,(%esp)
```

This instruction places the memory address 0x80484c4 onto the stack - the compiler just chose to use a different way of placing the memory address onto the stack than the standard push instruction. Note the parenthesis around *%esp* – this indicates a pointer. So, the mov command (AT&T syntax always

uses 'movl' instead of 'mov', but they are the same instructions) is actually placing the memory address into the address pointed to by the ESP register, not directly into the ESP register itself. Perhaps you noticed that in the prologue, eight bytes were reserved on the stack for local variables, even though no variables are defined in our source code. That was necessary in order to place the memory address on the stack in this manner. If those eight bytes had not been allocated, ESP would still be pointing to the same place as EBP, and the saved EBP value would be overwritten with the 0x080484c4 address (why the compiler uses this instead of a push instruction I don't know – that's up to the gcc developers :).

```
0x0804839b <main+23>: call 0x080482b0 <_init+56>
```

This is a call to a function at the address 0x080482b0. Since we have only one function that is called from our code, this must be the call to printf(). There was a push instruction (or the equivalent thereof) immediately before calling printf(), so that push must have placed an argument for printf() onto the stack. Our call to printf() only has one argument: the string to print. This can be double checked by examining the contents of 0x080484c4 (the address pushed onto the stack) by issuing the command:

```
(gdb)x/s 0x08048384  
0x08048384 <_IO_stdin_used+4>: "Hello World!\n"  
(gdb)
```

So, this is indeed our call to printf(), and our single argument, the “Hello World!\n” string, was appropriately placed on the stack just before it was called.

```
0x080483a0 <main+28>: mov $0x0,%eax
```

Remember that the EAX register holds any value that is returned by a function, and our main() function returns zero. So this instruction is placing the value 0 into EAX in preparation for a return.

```
0x080483a5 <main+33>: leave
```

The leave instruction cleans up the stack by removing all local variables from the stack and popping the saved EBP value off the stack into the EBP register, restoring it to its original value.

```
0x080483a6 <main+34>: ret
```

The ret instruction pops the top value off of the stack and places it into the EIP register. Since all data through the saved EBP value has been removed by the leave instruction, the top most piece of data on the stack is the saved EIP value; thus, the leave and ret instructions enable the function to properly return. Since these last three instructions (main+28 through main+34) prepare the function to return and clean up data placed on the stack by the prologue, they are known as the function's epilogue.

Here is the same disassembly, but this time printed in the Intel syntax, and commented for an easier feel of how the program flows:

```

0x8048384    push    ebp        <--- Save the EBP value on the stack
0x8048385    mov     ebp,esp    <--- Create a new EBP value for this function
0x8048387    sub     esp,0x8     <---Allocate 8 bytes on the stack for local variables
0x804838a    and     esp,0xffffffff <---Clear the last byte of the ESP register
0x804838d    mov     eax,0x0     <---Place a zero in the EAX register
0x8048392    sub     esp,eax     <---Subtract EAX (0) from the value in ESP
0x8048394    mov     DWORD PTR [esp],0x80484c4 <---Place our argument for the printf() function
                                                (at address 0x08048384) onto the stack
0x804839b    call    0x80482b0 <_init+56> <---Call printf()
0x80483a0    mov     eax,0x0     <---Put our return value (0) into EAX
0x80483a5    leave   <---Clean up the local variables and restore the EBP value
0x80483a6    ret     <---Pop the saved EIP value back into the EIP register

```

As you can see, they are all the same instructions, just formatted a little differently; note also how the Intel syntax indicates a pointer reference as opposed to the AT&T syntax.

### Disassembling Without The Source

Next, we will examine a program for which we have no source code, called helloworld2. We will attempt to reconstruct the original source code as closely as possible, and to understand how the program operates. Let's start out by running the program to see what it does:

```

$./helloworld2
Hello World!
$

```

So far, it appears no different than our first program. We know that it prints out a string to stdout, so it probably uses the printf() function. If we are disassembling this in Linux, we can use the strings command to look for the “Hello World!” string and anything else that may be interesting:

```

$strings helloworld2
/lib/ld-linux.so.2
__Jv_RegisterClasses
__gmon_start__
libc.so.6
printf
_IO_stdin_used
__libc_start_main
GLIBC_2.0
PTRh@
[^_]
Hello World!
Goodbye World!

```

We see our “Hello World!” string, but there is also a “printf” string (indicating that the program does indeed use printf), and another interesting string, “Goodbye World!”. Now, let's look at the main() function in gdb:

*(gdb) disassemble main*

*Dump of assembler code for function main:*

```
0x080483af <main+0>: push  %ebp
0x080483b0 <main+1>: mov   %esp,%ebp
0x080483b2 <main+3>: sub   $0x8,%esp
0x080483b5 <main+6>: and   $0xffffffff0,%esp
0x080483b8 <main+9>: mov   $0x0,%eax
0x080483bd <main+14>: sub   %eax,%esp
0x080483bf <main+16>: movl  $0x1,0x804961c
0x080483c9 <main+26>: call  0x8048384 <myprint>
0x080483ce <main+31>: mov   $0x0,%eax
0x080483d3 <main+36>: leave
0x080483d4 <main+37>: ret
```

Here we see the same prologue as before between main+0 and main+14. However, at main+16 we see that the number 1 is being moved into the memory address at 0x0804961c. This memory address is referenced directly, not as an offset from EBP, indicating that it is a global, not local, variable. Since the number 1 is being moved into it, it is safe to assume that this is an integer variable as well; we will call it var1. Next is a call to a function named 'myprint', which takes no arguments. Immediately afterwards we see the epilogue where 0 is moved into EAX, and leave and ret are called. So we now know that the main function simply sets a global integer variable to 1, calls a second function, then returns zero. We can reconstruct the main() function's source code to read:

```
int var1; /* The global integer variable */

int main()
{
    var1 = 1;
    myprint();
    return 0;
}
```

Next, let's examine the myprint() function:

*(gdb) disassemble myprint*

*Dump of assembler code for function myprint:*

```
0x08048384 <myprint+0>: push  %ebp
0x08048385 <myprint+1>: mov   %esp,%ebp
0x08048387 <myprint+3>: sub   $0x8,%esp
0x0804838a <myprint+6>: cmpl  $0x1,0x804961c
0x08048391 <myprint+13>: jne   0x80483a1 <myprint+29>
0x08048393 <myprint+15>: movl  $0x80484f4,(%esp)
0x0804839a <myprint+22>: call  0x80482b0 <_init+56>
0x0804839f <myprint+27>: jmp   0x80483ad <myprint+41>
0x080483a1 <myprint+29>: movl  $0x8048502,(%esp)
0x080483a8 <myprint+36>: call  0x80482b0 <_init+56>
0x080483ad <myprint+41>: leave
0x080483ae <myprint+42>: ret
```

We see that after the prologue, at myprint+6, there is a comparison operation. It is comparing the value stored in 0x0804961c (var1, the global variable we saw in the main() function) with the number 1. Immediately afterwards is a jne instruction. So, if var1 is not equal to 1, the the program will jump down to myprint+29, but if it is equal to 1 (which we know it is, because it was set to 1 in the main() function), it will execute the next instruction at myprint+15. Since we know that the jump will not be taken, let's look at what happens at myprint+15.

Myprint+15 pushes the memory address of 0x080484f4 onto the stack (again, using the mov instruction instead of push, but achieving the same end result), then calls a function that is located at 0x080482b0. This means that the function at 0x080482b0 is passed one argument; let's take a look at what that argument is by examining what is stored at the address 0x080484f4:

```
(gdb)x/s 0x080484f4
0x080484f4 <_IO_stdin_used+4>: "Hello World!\n"
```

This is our "Hello World!" string, and since we know that printf() is being used to print it to stdout, then the function at 0x080482b0 must be printf(). After the call to printf(), the program jumps down to myprint+41, which begins the function's epilogue. Since no value is placed in EAX before returning, and we know that the main() function does not examine EAX or place it anywhere in memory after calling the myprint() function, we can surmise that this function doesn't return a value.

But let's now look at what would happen if var1, for some reason, did not equal one. The jne instruction specifies that the program would jump down to myprint+29, which places a memory address (0x08048502) onto the stack in the same manner as before, then calls a function at 0x080482b0 – the printf() function. This means that either way printf() is called, it is just provided with a different argument. Taking a look at the contents of 0x08048502, we see that this alternate argument is the "Goodbye World!" string that we saw with the strings command earlier:

```
(gdb)x/s 0x08048502
0x08048502 <_IO_stdin_used+18>: "Goodbye World!\n"
```

We now know enough about the myprint() function to reconstruct its original source code as well:

```
void myprint()
{
    if(var1 == 1){
        printf("Hello World!\n");
    } else {
        printf("Goodbye World!\n");
    }
}
```

While there is no real purpose of the if-else statement (since var1 will always be equal to zero), I wanted to include it in order to show what a conditional statement looked like in assembly code. It is very important that you are able to recognize and understand conditional statements in assembly, as more complex comparisons (such as long case/switch statements) will be more difficult to follow.

## **Conclusion**

This paper covered necessary background information and provided some simplistic examples in order to introduce the basic concepts of RCE. You should now have a good grasp of how to read and interpret disassembled code, identify variables and functions, and translate the assembly code back into a high-level language. In most cases however, you will be working with larger programs that are much more difficult to analyze; you may also be only interested in a particular part of the program, or you may want to examine all instances of a specific function. In the next paper, we will introduce some new tools and debugging techniques, as well as cover some more advanced RCE methods in order to deal with such situations.

## References

Wikipedia: [http://en.wikipedia.org/wiki/Reverse\\_engineering](http://en.wikipedia.org/wiki/Reverse_engineering)

Exploiting Software: <http://www.informit.com/articles/article.asp?p=353553&seqNum=2&rl=1>

AT&T vs Intel - [http://www.gnu.org/software/binutils/manual/gas-2.9.1/html\\_chapter/as\\_16.html](http://www.gnu.org/software/binutils/manual/gas-2.9.1/html_chapter/as_16.html)

Dev-C++ - <http://bloodshed.net/devcpp.html>