

# COMP 40 Assignment: Analysis of AMD64 Assembly Code

Assignment due Tuesday, November 1 at 11:59 PM.

## Contents

<b>1 Purpose and overview</b>	<b>1</b>
<b>2 Making a game of it: the binary bomb</b>	<b>2</b>
2.1 How to get a bomb . . . . .	2
2.2 Preparing to defuse your bomb . . . . .	3
2.3 Defusing the bomb . . . . .	3
<b>3 What we expect from you</b>	<b>4</b>
<b>4 Resources</b>	<b>5</b>
4.1 Tools . . . . .	5
4.1.1 The Data Display Debugger . . . . .	5
4.1.2 The GNU debugger . . . . .	6
4.1.3 The objdump program . . . . .	6
4.1.4 The nm program . . . . .	6
4.1.5 The lowly but useful strings program . . . . .	6
4.2 Documentation . . . . .	6
<b>5 How to analyze assembly code</b>	<b>8</b>
<b>6 Acknowledgements</b>	<b>9</b>

## 1 Purpose and overview

The purpose of this assignment is to take the first step toward one of the central goals of this course: to acquire a working knowledge of the AMD64 instruction-set architecture.<sup>1</sup> To that end you will read, analyze, and understand a half-dozen or so C procedures given access only to an executable binary. An important secondary objective is for you to learn to debug programs using the Data Display Debugger (DDD).

Fundamental questions about assembly code include

- When a procedure starts executing, where does it find its parameters, and where should it put its return value?
- What does each instruction do when executed?
- How are expressions translated into operations on memory, constants, and registers?
- How are statements like `if`, `while`, `for`, and `switch` translated into control flow?

The homework requires *analysis of assembly-language programs*. Important questions include

- What state of the machine determines whether a conditional branch is taken, and how did that state get set?
- What are the possible values of a machine register, and how did that value get set?
- What values could be returned from a procedure, and how are those values determined?

---

<sup>1</sup>Now that Intel also supports AMD64, it is more officially known as x86\_64, but certain old dogs have trouble learning new tricks, not to mention that credit should be given where credit is due. Down with revisionist history!

- What values could be input to a given procedure in order to create a desired return value?

You'll mix three kind of activities:

- Analyzing assembly code
- Using your analysis to defuse a “binary bomb”
- Using the results of your analysis to reconstruct C code

You'll hand in a written report of your analysis; your work on defusing the bomb is reported automatically.

## 2 Making a game of it: the binary bomb

A “binary bomb” is a program that consists of a sequence of phases, each of which can be *defused* by the proper input. In a COMP 40 bomb, each phase expects one line on standard input; if the phase receives the input it expects, it is defused, and you continue to the next phase. If the phase receives any other input, the bomb *explodes*: it prints

BOOM!!!

The bomb has blown up.

Worse, it sends a message to the class “bomb server,” and you you lose  $\frac{1}{40}$  of the available credit for that phase. Be careful!

The bomb is divided into six phases. If you defuse a phase successfully, you will receive either full credit (if your bomb has never blown up) or partial credit (if your bomb blew up at any point). If you fail to defuse a phase, you receive no credit for defusing. The phases get progressively harder to defuse, but the increasing difficulty should be offset by the expertise you gain as you move from phase to phase. The final phase will challenge even the best students—don't start at the last minute.

We've doctored the bomb to help you avoid repetitive work and slips of the finger. First, the bomb ignores blank input lines. Second, if you run your bomb with a command-line argument, for example,

```
./bomb inputs1-3.txt
```

then the bomb reads input lines from `inputs1-3.txt`, which might contain the inputs expected by phases 1 to 3. Afterward, when the bomb reaches end of file on `inputs1-3.txt`, it switches to reading standard input. Use this feature to your advantage by putting correct phrases into a file!

### 2.1 How to get a bomb

On this assignment you are encouraged to work in pairs. Each pair should use *one and only one* bomb. Bombs are available from <http://bomb.cs.tufts.edu:54321> (think countdown). This URL works *only from inside the firewall*; if you are not in the lab, `ssh` to `linux` and download your bomb with `elinks` or `w3m`. Each bomb is an executable binary which has been compiled from C; it is packaged in a tar file which can be unpacked with, e.g.,

```
tar xvf bomb4.tar
```

Save the `bombk.tar` file to the directory in which you plan to do your work. Unpack the tar file by

```
tar xvf bombk.tar
```

This command will create a directory called `./bombk` with three files:

- File `README` identifies the bomb and its owners.
- File `bomb` is the executable binary bomb.
- File `bomb.c` is the source code of the bomb's main routine, complete with snarky comments.

If you make a mistake, such as losing your bomb or typing in the wrong group members, request another bomb.

## 2.2 Preparing to defuse your bomb

Run `objdump -d bomb` to get a printout of the assembly code of your bomb. You'll be especially interested in functions `main` (to which you have the source) and functions `phase_1` through `phase_6`. You will also be interested in other functions, but anything before `main` in the printout is probably not interesting.

You must *defuse the bomb on linux.cs.tufts.edu or one of the machines in Halligan 116, 118, or 120*. If you wish to work remotely, you will need either an X windows client or a VNC client. If you are not already familiar with X clients and ssh forwarding, I recommend you use a VNC client, available from <http://www.realvnc.com>. Information about how to set up VNC is available at <http://www.cs.tufts.edu/comp/40/howto-vnc/>, with emphasis on using the Windows PuTTY client to get through the firewall. If you are using Linux you can SSH tunnel by using the `-via` option on the VNC viewer.

Connect to a class server, launch the Data Display Debugger (DDD) on your bomb, and start stepping through the code.

- You will almost certainly want to use the stop sign to set breakpoints to keep the bomb from executing more code than you intend; after all, if you get the input wrong and the bomb keeps executing, the bomb will explode!
- You will also find it useful to use the flashlight to watch the values of particular registers and stack locations. You can watch any C expression, including expressions involving registers. For example, the expression

```
((int *)($rbp + $rbx * 4))[0]
```

watches the value in the address known to the assembler as `0x0(%rbp,%rbx,4)`.<sup>2</sup>

## 2.3 Defusing the bomb

You will need access to volumes of information, including

- An architecture manual for AMD64 Manuals are available from AMD<sup>3</sup> and Intel<sup>4</sup>. I find the AMD manuals a bit easier to read, but the Intel manuals have a compensating advantage: Intel gives all the instructions in simple alphabetical order, in two volumes. If you open both Intel manuals in Adobe Reader, you can quickly look up any instruction using the alphabetical table of contents.
- Documentation of the *procedure calling convention*, which is also known as the Application Binary Interface (ABI)<sup>5</sup>
- Documentation for the GNU assembler
- Documentation for DDD<sup>6</sup>
- Possibly documentation for gdb

Please see the **Resources** section below. There is too much information, but the critical parts are

- Understanding how procedures' parameters are passed in registers and on the stack (ABI)
- Recognizing instructions and addressing modes
- Remembering that *the GNU tools put the destination on the right, but the manufacturers' documentation puts the destination on the left*<sup>7</sup>

---

<sup>2</sup>For reasons known only to the people at the Free Software Foundation, the assembly-language tools use the % sign to refer to a register name, but the debuggers use a \$ sign.

<sup>3</sup>See URL <http://developer.amd.com/documentation/guides/Pages/default.aspx#manuals>.

<sup>4</sup>See URL <http://www.intel.com/products/processor/manuals/index.htm>.

<sup>5</sup>See URL <http://www.cs.tufts.edu/comp/40/readings/amd64-abi.pdf>.

<sup>6</sup>See URL <http://www.gnu.org/manual/ddd/>.

<sup>7</sup>Someone must pay.

As for the rest, you are not expected to understand much of it—just use it judiciously to get through your bomb.

Once you are comfortably seated on this indigestible brick of information, launch DDD and start single-stepping. Key skills are

- Aborting execution if things don't go your way
- Bringing up the machine-code window and the command window
- Setting breakpoints
- Inspecting values in registers and memory
- Watching expressions involving registers and memory

Using the debugger is a critical skill; time invested now will be repaid manyfold throughout your career.

### 3 What we expect from you

Results from defusing the bomb are automatically sent to your instructor, so there is no need to hand them in. You should *validate your results* by checking the page at

<http://www.cs.tufts.edu/comp/40/bombstats.html>

This web page is updated frequently and shows everyone's progress.

In addition to defusing the bomb, we expect you to *write down the results of your analysis*:

- In clear, grammatical English, please explain what is happening in Phases 1 through 6. We want the simplest possible explanations; for example, you might say “Phase 1 reads two numbers and explodes unless the second is evenly divisible by the first.”<sup>8</sup> When you encounter more complex phases, feel free to support your explanations with pictures of stack-frame layout and register contents.
- For each phase, write C code which represents your best guess as to how the function might be implemented. This code must compile, but it need not run.
  - Please include code for each phase.
  - Please include code for each phase-specific helper function.
  - You need not include code for general-purpose helper functions whose contracts are obvious, such as `number_is_divisible_by_seven` or `read_eight_strings`. But for each such helper function, please *do* include a suitable `extern` declaration with the correct argument and result types.
  - Within an individual phase, if there is code that you can prove is not needed to defuse the bomb, you may omit it.

We therefore expect you to submit the following:

1. A README file which
  - Identifies you and your programming partner by name
  - Identifies your bomb by number
  - Acknowledges help and collaboration
  - Gives your informal explanations of Phases 1 through 6
  - Says approximately how many hours you have spent on the assignment

---

<sup>8</sup>Except it isn't true, so you wouldn't say it.

2. A defuse file that gives the input lines needed to defuse your bomb (for as many phases as you have successfully defused).
3. At your option, a file `pictures.pdf` containing diagrams that support your README file
4. A file `phases.c` which gives your notion of C code that is equivalent to the machine code in Phases 1 through 6

Hand-drawn pictures can be scanned into PDF using the machine in Halligan 102.

## 4 Resources

### 4.1 Tools

#### 4.1.1 The Data Display Debugger

The Data Display Debugger is actually a user interface that operates on top of gdb. Here are some things to know:

- You can start DDD on a local program by running

```
ddd bomb
```

- If you are on a machine on which DDD is installed, you can run DDD remotely by giving the name of a remote host and a path to a program. Here's an example:

```
ddd -rhost linux.cs.tufts.edu /h/nr/cs/40/arith-student/40image
```

- The blank box on upper left, just to the right of `() :`, is the *argument field*.
  - Clicking the white area to the left of a statement may put a *source-code location* in the argument field; you can then use the *stop sign* to set or clear a breakpoint. You can also type the name of a function into the argument field and use the stop sign to set a breakpoint there. (Function `phase_1` might be an interesting one to start with.)
  - You'll want to think more deeply about where to set breakpoints to keep your bomb from blowing up every time you type in a wrong input.
  - Clicking a variable or sweeping out an expression may put an *expression* in the argument field; you can then use the flashlight to put a display of that expression in the display window.
  - You can often double-click to get an immediate breakpoint or data display.
  - It can also be useful to type in or edit an expression of your own and to display it using the flashlight. Casts will play a critical role here; for example, if you think that register `%rax` points to an array of four unsigned integers, you might want to display each of these four expressions:<sup>9</sup>

```
((int *)$rax)[0]
((int *)$rax)[1]
((int *)$rax)[2]
((int *)$rax)[3]
```

Type an expression into the argument field and click the display flashlight, then edit the expression to show the next element. You can easily display all four at once.

- Another way to display a sequence of integers in memory is to use the Memory tool under the Data menu.

---

<sup>9</sup>This example should make it very clear that DDD is designed for debugging source code, *not* machine code. To look at a machine register, you have to remember to name it with a `$` sign, you have to cast it to some useful C type, and then finally you can use the results in an expression. Tedious.

- You will find it useful to turn on Display Machine Code under the Source menu.
- Unfortunately neither DDD nor gdb understands the assembly-language syntax, so when you see an assembly-language expression, you'll have to translate it to a C expression.
- The commands in the command-tool window are almost all inherited from gdb, so you would do well to print one of the one-page gdb quick-reference guides referred to in the next section.
- I recommend using Edit::Preferences to change default fonts to `helvetica-bold` and monospace fonts to `bitstream vera sans mono-medium`.
- The very bottom window, where gdb stuff is going on, is also where you have standard input and standard output, unless they are redirected using the “run with arguments” feature under the Program::Run... menu.

The full documentation for DDD may be found at <http://www.gnu.org/manual/ddd/>. Our class lab on DDD highlights some especially useful commands.

#### 4.1.2 The GNU debugger

DDD is actually “just” a user interface. To control the underlying program, DDD, relies on the the GNU debugger, gdb. gdb is a command-line debugger that supports a handful of languages and a great many target machines. With luck, you'll avoid most of gdb, but you'll want to keep in mind that the gdb window at the bottom of the DDD interface allows you to interact directly with gdb. All of DDD's abilities to single step, show data, set breakpoints, and so on, are also present in gdb.

Full documentation for gdb can be found at <http://sourceware.org/gdb/documentation>. It is probably more useful to browse the textbook site at <http://csapp.cs.cmu.edu/public/students.html>, which has a number of gdb resources. In particular, the “Quick GDB x86-64 reference<sup>10</sup>” contains a very short summary of commands, most of which are in common with DDD. This is the place to learn what “Stepi” and “Nexti” actually mean.

#### 4.1.3 The objdump program

The command `objdump -d` is invaluable to print assembly language for all of the code in the bomb. You can also just look at individual functions. Having a printed copy of the assembly code, which you can write on, is an invaluable tool to keep track of your growing understanding of the bomb.

#### 4.1.4 The nm program

Running `nm -p bomb` will show all the names defined and used in the bomb. These names include all of the bomb's functions and global variables as well as all the functions the bomb calls. You may learn something by looking at the function names!

You'll also see many undefined names which should look vaguely familiar; they refer to names defined in the GNU C Library. This library is loaded dynamically, after the bomb starts. (Dynamic loading is also responsible for most of the junk between `_init` and `_start`. These are functions that are called from the bomb, which in turn make indirect calls through something called the Global Offset Table. At this point, sensible people run screaming from the room.)

For more information, some people like `objdump -t`, but I prefer the simplicity of `nm`. Check out the man page!

#### 4.1.5 The lowly but useful strings program

It won't get you far, but

```
strings bomb
```

will display the printable strings in your bomb. Sometimes `strings` squeezes a surprising amount of information out of a reticent program. If nothing else, `strings` will teach you not to store your passwords in the clear!

<sup>10</sup>See URL <http://csapp.cs.cmu.edu/public/docs/GDB-commands-x86-64.txt>.

## 4.2 Documentation

You may find the following documentation useful:

- The GNU assembler manual is at <http://sourceware.org/binutils/docs/as>. This manual is utterly overwhelming, but you may find the machine-dependent section useful; AMD64 is treated as a member of the 80386 family ([http://sourceware.org/binutils/docs/as/i386\\_002dDependent.html](http://sourceware.org/binutils/docs/as/i386_002dDependent.html)).

The most important thing to know is that the GNU people put the destination on the right (AT&T syntax) where Intel and AMD put it on the left. If you are moved to mayhem, Richard Stallman is but a short train ride away...

Another useful thing to know is the meaning of the address syntax; here are the addressing modes you're most likely to encounter:

```
%rax          == $r[0]    # similarly for other registers
%eax          == least significant 32 bits of %rax
              (presence may signal a 32-bit operation)
disp(base, index, scale) == $m[base + index * scale + disp]
disp(base)    == $m[base + disp]
label(,index,scale) == $m[label + index * scale]
symbol(%rip)  == relative reference to symbol in memory
```

- Documentation of the instruction-set architecture is available from both AMD ([http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30\\_2252\\_875\\_7044,00.html](http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_875_7044,00.html)) and from Intel (<http://www.intel.com/products/processor/manuals/index.htm>). People I know who have tried both prefer AMD's documentation.
- Sections 3.1–3.7 of Bryant and O'Hallaron explain assembly-language programming in detail. (You may also find Sections 3.8–3.10 useful for a review of machine-level representations of data; in the first edition, the same material is spread out over Sections 3.8–3.11.) Unfortunately, the explanations are written in terms of the 32-bit Pentium instruction set. The new edition of Bryant and O'Hallaron covers AMD64 in Section 3.13, but their approach is to describe the differences between AMD64 and IA-32 (the official name for the Pentium family architecture), which in your position I might find annoying. I don't know of a book available that treats the AMD64 instruction set as primary; if you do, please let me know.
- If you're using the first edition of Bryant and O'Hallaron, they have written a supplement for students using AMD64 machines<sup>11</sup>. Like the new chapter in the second edition, the supplement focuses on differences between AMD64 and IA-32. But you might find that the information is more digestible than the same information in raw form from the System V ABI.
- The System V ABI for AMD64<sup>12</sup> is long, but it contains a few chunks of very good information:
  - Page 12 gives the sizes and alignments of basic data types.<sup>13</sup>
  - Section 3.2 on pages 14–22 explains the calling convention. As is typical for a modern ABI, it is overly complex. The main points are as follows:
    - \* Stack-frame layout is sketched in Figure 3.3 on page 16.
    - \* Integer arguments are passed in integer registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9.
    - \* Floating-point arguments are passed in registers %xmm0 through %xmm7. Confusingly enough the %xmm*k* registers are called the SSE registers.<sup>14</sup>

<sup>11</sup>See URL <http://www.cs.tufts.edu/comp/40/readings/asm64-handout.pdf>.

<sup>12</sup>See URL <http://www.cs.tufts.edu/comp/40/readings/amd64-abi.pdf>.

<sup>13</sup>All page numbers refer to the draft version 0.99 dated December 7, 2007.

<sup>14</sup>Shockingly, this nomenclature is *not* Richard Stallman's fault.

- \* As described on page 20, values are typically returned in `%rax` (integer) or `%xmm0` (floating-point), but for large return values, more complicated rules obtain. Especially noteworthy is the item 2 which normally applies to functions returning `structs`.
- \* Figure 3.4 on page 21 provides a very nice summary of how registers may be used and which are preserved by function calls.

The rest of this very large document can be ignored.

## 5 How to analyze assembly code

The primary technique involved is to *maintain at each program point an account of the contents of machine registers and the procedure's stack frame*.<sup>15</sup> Registers and stack slots that you don't care about can be omitted from description or can be written using the “don't care” value  $\perp$  (pronounced “undefined”). The technique is best applied both *backwards* and *forwards*:

- *Backwards dataflow analysis* considers the desired machine state *following* an instruction and asks what state is necessary to achieve that state *preceding* an instruction. For example, if the desired state following the instruction involves having register `%rbx` greater than 5, and the instruction in question is

```
add    $0x1,%rbx    # %rbx := %rbx + 1
```

then the desired state preceding the instruction must be to have  $\%rbx + 1 > 5$ , or equivalently,  $\%rbx > 4$ , or equivalently. The condition characterizing the desired state preceding the instruction is called the *weakest precondition*, and I calculated it by substituting the right-hand side of the assignment for the register on the left.

- *Backwards control-flow analysis* considers what has to be done to reach (or to avoid) a particular program point. For example, consider the following disassembled code:

```
40122d:    je      401234 <phase_1+0x17>
40122f:    callq   40155b <explode_bomb>
401234:    add     $0x8,%rsp
```

You might naturally wish to avoid ever having control arrive at address `0x40122f`, which calls the `explode_bomb` function. It is therefore natural to ask what has to be done to make sure the preceding condition jump *succeeds* in jumping around the call to the `add` instruction. Looking backward, we see the preceding instruction is

```
40122b:    test    %eax,%eax
```

Looking up this instruction in the architecture manual, we see that the instruction computes the “bitwise and” of `%eax` with itself (that is, the “bitwise and” of the least significant 32 bits of `%rax`), uses the result to set the sign, parity, and zero bit in the condition codes, and then discards the result. So the `je` instruction is mnemonic for “jump on equal,” which is the same as “jump if zero”,<sup>16</sup> which means the jump will succeed if and only if `%eax` is zero.

Now, as is utterly typical, the control-flow program has become a dataflow problem, and we must look backward to see what sets `%eax`. We haven't far to look, as the preceding instruction is

```
401226:    callq   40125f <strings_not_equal>
```

and we know from the calling convention that a 32-bit integer value is returned in `%eax`. Thus it is safe to conclude that in order to prevent the bomb from exploding, the `strings_not_equal` function must return zero. Suggestive, is it not?

<sup>15</sup>You are not yet expected to know what a stack frame is; it is covered in your book and will be covered in class.

<sup>16</sup>You can see where this analysis is going to get tedious. And that you're going to need to start early.



Backwards analysis is what Sherlock Holmes used to love: reasoning from effects to what causes must produce those effects. But to validate your reasoning, you'll want to indulge in a little forwards analysis. This is sometimes called "symbolic execution."

- *Forwards dataflow analysis* asks if a register or stack slot contains a certain value, how will that value affect subsequent execution?
- *Forwards control-flow analysis* asks if a register, stack slot, or condition-code flag contains a certain value, what instructions can be reached in a subsequent execution.

You can do forward analysis in your head, but it's much easier to let DDD do it for you. You can set a breakpoint anywhere you like, use DDD to *change the values in registers and memory*, then single-step forward to see what happens. Just don't go too far, or you might blow yourself up!

## 6 Acknowledgements

The "binary bomb" idea and the associated code were developed by David O'Hallaron, who has a twisted sense of humor.<sup>17</sup> Many of the informational resources were suggested by Soha Hassoun. Backwards dataflow analysis was invented by Sherlock Holmes, who was invented by Arthur Conan Doyle.

---

<sup>17</sup>Terry Pratchett helped with the footnotes.