CS244, STELLENBOSCH UNIVERSITY

TUT 7: THE BOMB LAB

A2 PREPARATION, 2021

Giving up on assembly language was the apple in our Garden of Eden. Languages whose use squanders machine cycles are sinful.

—Alan J. Perlis, *Epigrams on Programming*

YOUR MISSION

You will receive, via e-mail, a 32-bit Linux executable file, which was hardcoded to your student number. No two are alike, and in particular, they differ in the way the various "solutions" are ordered into phases. Once the bomb is running, it requires keys—passwords or passphrases—to be defused successfully. These key sequences are unique to your bomb. When attempting to defuse a particular phase, all preceding keys must be entered again to reach this phase. If any password or passphrase is incorrect, the bomb "explodes", in the process contacting a server to record the event.

THE TOOLS OF THE TRADE

The bomb executable has been stripped of debugging symbols, but the names of the underlying functions and subroutines have been left intact. The object of this exercise is learning to use the GNU Debugger and other binary utilities on the Linux platform.

GDB GDB, the GNU Debugger[1], is a standard tool, useful not only for debugging, but also for reverse engineering. At the very least, you should learn how to set breakpoints, and how to step through the individual instructions in an executable.

The manual is available as a PDF on the course website. There is also a `gdbinit` file you should use; download and save it as `~/.gdbinit`, that is to say, in your home directory. It is especially helpful to position the various components of the text user interface so that you can see the code through which you are stepping. It also defines the following commands and shortcuts:

1. `s` – to *step* through the file, a single instruction at a time;
2. `n` – to step to the *next* line of the source file, but any functions are executed without stopping;
3. `showbase` – show the context (the surrounding area) of the variables pushed on top of the stack before a function call; and
4. `showstack` – show the context of the current top of the stack.

Finally, the `gdbinit` file has some examples that show how to set breakpoints.

[1]For more detailed information, refer to `https://www.gnu.org/software/gdb/`. You should also seriously consider installing GEF; see `https://github.com/hugsy/gef`.

GNU BINUTILS My preferred way of starting analysis of any unknown Linux executable to use is the GNU binary utilities[2]. As a matter of opinion, I find it easier to try and understand programs in a static context (examining the disassembled code and core dumps), and not a dynamic context (stepping through a running program with a debugger). That is, I prefer to examine the executable immobilised and docile, as opposed to on its hind legs, rearing and breathing fire.

NM The `nm` program lists all symbols from object files. These symbols include the names the programmer defines, for example, the names of functions. The bomb has been compiled from C source code and NASM assembly code. Therefore, you should be on the lookout for the `main` and `_start` symbols.

OBJDUMP The `objdump` program allows you to disassemble an object file. The following invocation should be particularly useful:

```
$ objdump -M intel -D bomb
```

Since GDB can display limited context on the available screen real estate, and since this context is tied to the current instruction being executed, I find it useful to redirect the `objdump` output to a text file so that I can have both the debugger and object dump open at the same time.

OTHER BINUTILS The `strings` program lists all string literals in an ELF file. The `size` program can list the sections of an object file. The `readelf` program displays information for any ELF file.

RULES

There really are no rules, except to note that the project is not on network security or social engineering. To explain by example:

1. The bombs are only guaranteed to work on the NARGA machine. You may work from outside the university network, provided your bomb can reach the server. Otherwise, it will refuse to run. Even if you do find a workaround to this restriction, remember the bomb eventually does have to access the server, to register your defusions.

2. You MAY use disassembly tools, *but you MUST NOT use any decompiler.* Should I detect use of decompilers, those involved will be penalised.

3. Hacking the protocol is frowned upon, because this is not a network course. The protocol is protected to some extent, and your solutions are checked again by the server. Should I detect protocol hacking going on, you will be penalised by having lots of marks deducted.

4. Writing a drop-in local replacement for the server is border-line, once again, because this is not a network course. It might yield some results, but will not work for all phases. I think it is a waste of time, considering the time you do have.

[2] For detailed information, head to `https://www.gnu.org/software/binutils/`.

5. Hacking the bomb executable itself is allowed. Just remember that, in the end, you have to run the bomb unaltered so that the server can register your solving the phases.

I suggest the following approach:

1. Analyse the bomb executable with the GNU binutils to discover its secrets.

2. To register your solutions, run the bomb in GDB, to try and protect against accidentally triggering an explosion.

MARKING SCHEME

As communicated at the beginning of the semester, this tutorial will be examined as part of the A2 assessment opportunity. When you solve a particular phase successfully, the server will reply with a secret key that you must (1) write down, and (2) bring to the A2 examination venue.

The marking scheme is pass/fail per phase solved. The weighting for each phase is given in the table below.

PHASE	WEIGHT	CUMULATIVE
1	25%	25%
2	25%	50%
3	15%	65%
4	15%	80%
5	20%	100%