# DD2460 – Graded Lab Assignment
# Memory Safety

## 1 Overview

In this assignment, you will analyze *chibicc*, a tiny C compiler.

**Description:** *chibicc* is a program that converts source code written in the C language into assembly language, that is, a C compiler. The compiler itself is also developed using C.

**URL:** https://github.com/cesarsotovalero/chibicc

**Note:** Please do not download the source from the official repository, as it does not contains the injected fault, so there would be not much to see.

### 1.1 Installation and example

1. Unpack the sources:
```
$ git clone https://github.com/cesarsotovalero/chibicc
$ cd chibicc
```
2. Compile the sources:
```
$ make
```
3. Run the tool for the *nqueens* example:
```
$ chmod +x chibicc
$ ./chibicc examples/nqueens.c > tmp.s
```
4. (Later:) Compile the generated intermediate code:
```
$ gcc -static -o tmp tmp.s
```
5. (Later:) Run the example:
```
$ chmod +x tmp
$ ./tmp
```

**Note:** In the given version, `chibicc` will crash and leave an empty file. The last two steps will only run successfully on Linux *after* you have solved the assignment. They are not required for this exercise. You can also execute the script `run.sh`, which executes all the previous tasks.

## 2 Valgrind usage

Install valgrind (either as a package for your OS, or by downloading it from https://valgrind.org/). Run Valgrind on the binary with the *nqueens* example as input:

```
$ valgrind --leak-check=full ./chibicc examples/nqueens.c > tmp.s
```

Valgrind will help you to understand why the `chibicc` crashes. You will see an error trace similar to this one:

```
==95176== Memcheck, a memory error detector
==95176== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==95176== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==95176== Command:  ./chibicc examples/nqueens.c
==95176==
--95176-- run:  /usr/bin/dsymutil "./chibicc"
==95176== valgrind:  Unrecognised instruction at address 0x1002cf4db.
==95176== at 0x1002CF4DB: __abort (in /usr/lib/system/libsystem_c.dylib)
==95176== by 0x1002CFD7D: __stack_chk_fail (in /usr/lib/system/libsystem_c.dylib)
==95176== by 0x10000940B: read_string_literal (tokenize.c:0)
==95176== by 0x10000902B: tokenize (tokenize.c:302)
==95176== by 0x1003052FF: ???  (in /dev/ttys007)
==95176== by 0x428:  ???
==95176== Your program just tried to execute an instruction that Valgrind
==95176== did not recognise.  There are two possible reasons for this.
==95176== 1. Your program has a bug and erroneously jumped to a non-code
==95176==    location.  If you are running Memcheck and you just saw a
==95176==    warning about a bad jump, it's probably your program's fault.
==95176== 2. The instruction is legitimate but Valgrind doesn't handle it,
==95176==    i.e.  it's Valgrind's fault.  If you think this is the case or
==95176==    you are not sure, please let us know and we'll try to fix it.
==95176== Either way, Valgrind will now raise a SIGILL signal which will
==95176== probably kill your program.
==95176==
==95176== Process terminating with default action of signal 4 (SIGILL)
==95176== Illegal opcode at address 0x1002CF4DB
==95176== at 0x1002CF4DB: __abort (in /usr/lib/system/libsystem_c.dylib)
==95176== by 0x1002CFD7D: __stack_chk_fail (in /usr/lib/system/libsystem_c.dylib)
==95176== by 0x10000940B: read_string_literal (tokenize.c:0)
==95176== by 0x10000902B: tokenize (tokenize.c:302)
==95176== by 0x1003052FF: ???  (in /dev/ttys007)
==95176== by 0x428:  ???
```

## 3 Explanation

There is an operation violating memory safety (in a serious way) inside `read_string_literal`.

Open the file `tokenize.c` and go to `read_string_literal` (line 180).

```
static Token *read_string_literal(Token *cur, char *start) {
  char *p = start + 1;
  char buf[32];
  int len = 0;

  for (;;) {
    if (*p == '\0')
      error_at(start, "unclosed string literal");
    if (*p == '"')
      break;

    if (*p == '\\') {
      p++;
      buf[len++] = get_escape_char(*p++);
    } else {
      buf[len++] = *p++;
    }
  }
  ...
```

You can perhaps guess the meaning of the code of the `for` loop: It tries to read a string literal that has started with a double quote (`"`) and exits when the closing quote is found. The loop also handles the backslash to escape the character following it, and copies over the resulting literal into the buffer `buf`. The buffer has a fixed length and is allocated on the stack.

## 4 Mandatory task: Error analysis

1. Explain the memory layout of the stack-allocated variables, the frame pointer, and the return pointer, in `read_string_literal`. Use the lecture slides from the memory safety lecture as a guide.
2. Explain what happened and why valgrind stops the program here.
3. Why is this potentially dangerous when running in an environment without memory safety checking?

## 5 Optional tasks

### 5.1 Fix (+0.5 points)

Observe that the size of `buf`, given by `sizeof(buf)`, is fixed. Access is controlled by `len`.

**Deliverable**: Write a patch that fixes the memory safety issue, using `error_at` to abort parsing when necessary.

### 5.2 Fuzzing (+0.5 points)

1. Clone radamsa from its home page: https://gitlab.com/akihe/radamsa

   ```
   git clone https://gitlab.com/akihe/radamsa
   cd radamsa
   make
   ```

   You do not need to run `make install`.

2. Set the environment variable `RADAMSA` so it points to the executable. Example:

   ```
   export RADAMSA=$PWD/bin/radamsa
   ```

3. Change back to the location where you have `chibicc`.

4. Create a variant of the example input program *without* the long string:

   ```
   grep -v Memory examples/nqueens.c > examples/base.c
   ```

5. Use fuzzing on the base example and inspect the result and/or run valgrind as above. You can create the example as follows:

   ```
   $RADAMSA examples/base.c > examples/fuzzed.c
   ```

   Try this step ten times and look into what types of perturbations you get.

**Deliverable**: Comment on your results:

1. What types of changes does `radamsa` introduce?

2. How many times has a string literal been changed?

3. Are the odds of getting a string literal that results in a buffer overflow any good?