This homework is focused on compilers and security, and links directly the resources that have been used in the related lecture sessions.

1 Warm-Up

1. Take a minute to read about Compilers bootstrapping, and in particular review the methods of "cross compiling", "hand compiling" and using an interpreter written in a different language.

2. Read about Quines and consult some of them on-line. I recommend carefuly going through the explanations shared at `https://stackoverflow.com/a/33535216` to understand how the following (variation of the presented) source code `self.c` can replicate itself:

```
#include <stdio.h>
char*s="#include <stdio.h>%cchar*s=%c%s%c;%cint
 ↪  main(void){printf(s,10,34,s,34,10,10);}%c";
int main(void){printf(s,10,34,s,34,10,10);}
```

   Compile it with e.g.

```
gcc -O3 -g -std=c99 -Wall -Wextra -Wmissing-prototypes -Wstrict-prototypes
 ↪ -Wold-style-definition self.c
```

   and observe the magic happens! If you *really* want to have your mind blown, I strongly recommend that you have a look at `https://github.com/mame/quine-relay/`.

2 Optimization Gone Wrong

Let me clarify the "msize" example that is presented at `https://www.redhat.com/en/blog/security-flaws-caused-compiler-optimizations` but actually introduced in `https://people.csail.mit.edu/nickolai/papers/wang-undef-2012-08-21.pdf`. It goes as follows: we *want* to raise an error if a variable `msize` is set to 0. Not knowing any better, we find the following code `opt_wrong.c` good enough:

```
#include <stdio.h>

int main() {
    int msize = 0;
    if(!msize)
    {
        printf("Msize is %d, so this will crash.\n", msize);
        msize = 1 / msize; /* Causes a division by 0 exception. */
    }
    return 0;
}
```

If that code is compiled with clang (version 13) and the level 1 of optimization (`clang -O1 opt_wrong.c`), the executable will simply display "Msize is 0, so this will crash." and *not* crash. Compiling it with all the optimizations off (with the `-O0` flag) results in a binary that triggers the exception.

You can inspect the .ll representation of both versions to realize that the "poor man's safety check" we tried to implement was simply removed by higher levels of optimizations of clang!

If, for some reason, you cannot reproduce this bug with your version of llvm, you can use the on-line demo on godbolt, that uses an old version of clang.

3 Trusting Source Code

3.1 Panorama

Using the discussions at `https://softwareengineering.stackexchange.com/q/287606/` and `https://security.stackexchange.com/q/138881`, briefly explain:

1. What makefiles are, and how they could be used to attack your system.

2. Why it can be extremely difficult to understand what a piece of code is actually doing (once again, I encourage you to have a look at the International Obfuscated C Code Contest and the Underhanded C Contest).
3. How Matt Godbolt managed to mitigates the risk of hosting an on-line compiler.
4. And, finaly, take a minute to be amazed by the compiler bombs that can e.g. produce a 16 GB file using 29 bytes of C code or run for a *long* time.

3.2 Trojan Source

This problem encourages you to understand the "Trojan Source" attack, that say that you *really* should not trust random code found on the Internet.

1. Read the first paragraph of "Bidirectional Text Spoofing" and briefly explain what "bidi" means.
2. Explain what "software supply chains" means, using search engines if needed.
3. Visit `https://trojansource.codes/` and read the whole page.
4. Confirm that at least two of the vulnerabilities demonstrated at `https://github.com/nickboucher/trojan-source` are still applicable to the compiler of your choice: select a programming language, an example, and empirically confirm that e.g. gcc or llvm is still vulnerable to it.
5. Note that github preemptively displays a warning message for the "Commenting out" attack, but not for the "homoglyph-function": can you explain why?
6. Read Section F., "Defenses", of the original paper at `https://trojansource.codes/trojan-source.pdf`.
7. Observe the "solution" offered by Atlassian: does it sound convincing?
8. Finally, observe that patches for llvm and gcc are currently developed and soon to be implemented. How much time did it take between the disclosure of the bug and this patch?