CSE127: Computer Security **Winter 2026**

PA1 — GDB + x86

Project Release: January 6, 2026 at 12:00:00 PT
Deadline: January 15, 2026 by 11:59:59pm PT

Introduction

The goal of this assignment is to become familiar with the setup that will be used for future assignments, such as the use of a virtual machine and the included turn-in script, as well as the basics of working with `gdb` and writing programs in `x86` assembly.

Forming a Group

This is a group project; you can work in a team of size at most two and submit one project per team. You are not required to work with the same partner on every project. You and your partner should collaborate closely on each part.

The code and other answers you submit must be entirely your team's own work. You may discuss the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone other than your partner. You may consult published references, provided that you appropriately cite them (e.g., with program comments).

Solutions must be submitted to Gradescope.

Getting Started

To complete this assignment, you will be provided with a VirtualBox VM pre-populated with the assignment files.

VM Image

In order to match the environment in which your submission will be graded, all work for this assignment must be done on the VirtualBox VM we provide, named pa1box. You can download the VM image here.

The VM is configured with two users: `student` (note that the user name is student instead of your school user name), with password `hacktheplanet`; and `root`, with password `hackallthethings`. The VM is configured with SSH on port 2222. Please note that SSH is disabled for root, so you can only SSH in as the student user. You can still log in as root using `su` or by logging into the VM directly.

To SSH into the VM:

```
ssh -p 2222 student@127.0.0.1
```

To copy files from your computer to the VM:

```
scp -P 2222 -r /path/to/files/ student@127.0.0.1:/home/student
```

To copy files from the VM to your computer:

```
scp -P 2222 student@127.0.0.1:/path/to/files/ /destination/path
```

Apple Silicon Setup

VirtualBox doesn't work on Apple Silicon machines. To use the vm image on these machines, you will have to install `qemu` and run the VM with it.

```
brew install qemu

qemu-system-x86_64 \
-cpu qemu64 \
-m 4096 \
-device e1000,netdev=net0,mac=08:00:27:D3:23:52 \
-netdev user,id=net0,hostfwd=tcp::2222-:22 \
-drive file=/Path/to/pa1box/pa1box-disk1.vmdk
```

User and ssh settings are the same as using VirtualBox.

Part 1: Using GDB (10 points)

Alice is a security engineer in a company called Security4All. She felt that their programs are running a bit weird recently. After doing some research, she suspect that the C compiler they are using might have been backdoored. It is a huge issue because a compromised compiler will potentially defeat all security mechanism they implemented in a higher level language. Now Alice heard that you have experience with `gdb` in some classes and asks for your help to learn the behavior of a program using `gdb`.

Files for this sub-assignment are located in the gdb subdirectory of the student user's home directory in the VM image; that is, `/home/student/gdb`. SSH into the VM and `cd` into that directory to begin working on it.

Inside the `gdb` directory, you'll find `fib.c`, a C program demonstrating the Fibonacci sequence; a `Makefile`; and `hw1.txt`, in which you'll record your responses to the questions below. The first step is to compile `fib` by running `make` on the command line.

To run the `fib` executable in GDB, run `gdb fib`. I recommend the following workflow in GDB:

1. **Starting.** Set breakpoints that you can use for later analysis:
   - `b foo` — break at function `foo`
   - `b 0x08048489` — break at the instruction at address 0x08048489
   - `r` — run the executable
2. **Analyzing.** Examine memory, registers, etc; disassemble code; show stack frames, backtrace, etc; and more:
   - `disas foo` — disassemble function `foo`
   - `i r` — view registers
   - `where` — view stack frames
   - `x <loc>` — examine memory
   - `x $eip` — examine current instruction pointer
   - `x /10x $esp` — examine 10 words at top of stack
   - `x /10x buf` — examine 10 words in `buf`
   - `x /10i $eip` — examine 10 instructions starting at instruction pointer
   - `x /10i foo` — examine 10 instructions starting at `foo`
3. **Continuing.** Continue analysis:
   - `c` — continue execution until next breakpoint/watchpoint
   - `si` — step to the next instruction
   - `s` — step to the next line of source code

Note that this is only a cursory overview of GDB; much more info is available from online resources.

Assignment Instructions

Complete the following exercises and fill out `hw1.txt` with your answers. Follow the directions in the template, **do not delete the square brackets!**

1. What is the value, in hex, of the `ecx` register when the function `f` is called? *(2 pts)*
2. Which register stores the value of the variable `i` in the function `main`? *(2 pts)*
3. What is the address, in hex, of the function `f`? *(2 pts)*
4. What is the name of the 6th instruction of the function `f`? *(2 pts)*
5. When `f` completes after being called from `main`, to which address in `main` does control return? Write your answer in hex form. *(2 pts)*

Submission

Submit `hw1.txt` to "Assignment 1 - GDB" on Gradescope. Gradescope will check that it has successfully detected your answers, but it will not give you your grade until the due date.

Part 2: `echo` in `x86` (10 points)

Bob is a developer at Security4All. He received an internal warning that the compiler might be compromised. However, not being able to use a compiler blocks the development of their project, which has to be released in a week. While the security team is still investigating the compiler, Bob decides to finish up development using raw assembly. However, it has been five years since Bob took his undergrad class about assembly, so Bob needs your help to write a simple x86 assembly program to refresh the knowledge.

Files for this sub-assignment are located in the `x86` subdirectory of the student user's home directory in the VM image; that is, `/home/student/x86`. SSH into the VM and `cd` into that directory to begin working on it.

For this part, you will be implementing a simplified version of the familiar `echo` command using raw x86 assembly code. The goal of this assignment is to familiarize you with writing programs directly in x86.

Note—this code should be written in x86 32-bit using AT&T syntax; other formats will not be accepted.

Your echo command must behave as follows:

- **When run with a single command line argument** (e.g., `./echo Hello`):
  1. Prints that argument back to the console's standard output (`stdout`).
  2. Prints a trailing newline (`\n`).
  3. Exits with code 0.
- **When run with too few command line arguments** (e.g., `./echo`) **or too many** (e.g., `./echo Hello World`):
  1. Prints *exactly* the error message `This command expects exactly one argument.` followed by a trailing newline (`\n`) to the console's standard error (`stderr`).
  2. Exits with code 1.

Your code should be written in the file `echo.s` inside the `x86` directory. A heavily commented example `echo.s` is provided, which simply prints the message `Hello World` (followed by a trailing newline) to `stdout`. Your job is to modify this program to meet the specification for `echo` above.

A `Makefile` is included, so you can build the echo binary by running make from the command line.

Submission

Submit echo.s to "Assignment 1: x86 echo" on Gradescope.

Part 3: AI Attestation

Submission

Submit your attestation to using AI on this PA (Yes or No) to "Assignment 1: AI Attestation" on Gradescope.

Helpful Hints

- In a Linux program, stdout is file descriptor number 1 and stderr is number 2.
- Linux programs start with argc at the top of the stack, accessible at `0(%esp)` from `x86`-assembly programs. Below it is `argv`, the array of pointers to (null-terminated) strings passed into the program as arguments. So `argv[0]` can be accessed from `x86`-assembly programs at `4(%esp)`, `argv[1]` at `8(%esp)`, `argv[2]` at `12(%esp)`, and so on.
- `argv[0]` is the name of the program executed, not the first argument, which is `argv[1]` if supplied (or `8(%esp)` for our purposes). Then `argc` (or `0(%esp)`) will be 1 if the program was not passed any arguments, 2 if it was passed 1 argument, and so on.
- This table of Linux system calls may come in handy.
- For people with AMD CPUs, you may find that your VMs don't boot unless you boot with sysvinit instead of systemd. This is normal, and will not affect your assignment.