CS 426: Computer Security
Fall 2025
Homework 2: Basic Buffer Overflow Attacks

Due Sept. 12th, 2025 @ 5:00 PM

1 Overview

The goal of this assignment is to help you gain hands-on experience on buffer overflow attacks. Before you start, you might want to go over Aleph One's seminal article *Smashing the Stack for Fun and Profit* available at this link `https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf` [One96]. You can also review the example we saw in class (or watch the recording of) on Thu, Sep 4, and attend the lecture on Tue Sep 9 (we are going to see more examples).

2 Environment Setup

Use the provided Kali Linux VM for this assignment. Buffer overflow exploitation is sometimes delicate, and their success depends on specific details of the target system, so you will be using the same Kali Linux VM used in your previous homework to develop and test your attacks. This will also promote a consistent experience for everyone in the class and facilitate the grading process. We have slightly tweaked the configuration to disable some of the security features that are commonly used in the wild but would complicate our buffer overflow attacks. We will use this precise configuration to grade your submissions, so do not use your own VM instead. You must download the targets into a folder owned by the VM; running targets in a shared folder might cause incorrect behavior. All materials for this question are available via Brightspace.

If you have not installed the VM, the image (`cs426-kali.ova`) file can be downloaded from this link: `https://app.box.com/s/x8hcgawqvdy9qipwca54sw7gxmsp3tgi`.[1] See the Homework 1 document for the installation details.

You must not attack anyone else's system without authorization! This assignment asks you to develop attacks and test them in a virtual machine you control. Attempting the same kinds of attacks on others' systems without authorization is prohibited by law and university policies and may result in *fines, expulsion, and jail time.* Per the course ethics policy, you are required to respect the privacy and property rights of others at all times, *or else you will fail the course.*

[1] This file is quite large (2 GB), so we recommend that you do not download it with your mobile data.

3 Assignment Setup

Files for the assignment are provided in Brightspace. Run the VM, download the file `hw2.tar.gz`, and place it under `/home/cs426/`. Then, open the terminal and navigate to `/home/cs426/` to unpack the package and compile the vulnerables. For these steps, run the commands below from your terminal.

1. Inside Kali VM, download the `hw2.tar.gz` package and unpack it.

   ```
   cd /home/cs426/
   wget -O hw2.tar.gz \
       https://app.box.com/shared/static/fhce00j3munm08kmo2t9vw71y1rqhexi.gz
   tar zxvf hw2.tar.gz
   cd hw2
   ```

2. Compile the randomized target binaries. First, set a cookie using your Purdue Career username (e.g., `./setcookie ling102`). Then, generate the randomized target source codes based on the cookie, and compile them into binaries:

   ```
   ./setcookie <username>           # replace <username> with your Purdue username
   make generate
   echo "cs426" | sudo -S make
   ```

Now you can start attacking the target binaries! Here are some requirements and hints that will help you in this assignment:

- **No specialized attack tools!** You may not use special-purpose tools meant for testing the security or exploiting vulnerabilities. You must complete the problem using only general purpose tools, such as `gdb`.
- **GDB:** You will find the GDB debugger extremely useful for finding buffer overflow exploits. Make use of your prior experience from Homework 1, and do not be afraid to experiment! This quick reference may also be useful: https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf.
- **x86 Assembly:** There are many good references for x86 assembly language but note that this assignment targets the 32-bit x86 ISA. The stack is organized differently in x86 and x86_64. If you are reading any online documentation, ensure that it is based on the x86 architecture, e.g., https://cs.dartmouth.edu/~sergey/cs258/tiny-guide-to-x86-assembly.pdf.

3.1 Miscellaneous

You may find it helpful to write an additional program or script that will assist in figuring out the correct input to exploit any of the vulnerabilities. Note that this can be separate from the script/program you will write to exploit the vulnerabilities. If you are unsure about what is acceptable for this assignment, please do not hesitate to contact us. Be sure to turn in the code for any additional program or script that you write.

4 Targets

This assignment requires you to attack four vulnerable C programs by identifying and exploiting their vulnerabilities. You will do this by writing four Python3 scripts, one for each target program. These scripts will provide malicious inputs or command-line arguments to exploit the vulnerabilities. You must use the default Python3 version provided in the Kali VM. Do not change the Python3 version or modify the script links.

IMPORTANT: Before attacks, make sure you have generated the `cookie` with your Purdue username, and compiled your target binaries with your `cookie` as in Section 3. Otherwise, we cannot grade your work. You can verify your cookie by uploading it to Gradescope under Homework 2 (Code).

4.1 vulnerable1: Overwriting a variable on the stack (10 points)

This program prompts for a username from `stdin` and prints a message:

```
Hi <username>!  Your grade is F.
```

Your task is to provide an input that causes the program to output:

```
Hi <username>!  Your grade is A+.
```

where `<username>` here is your Purdue username.

To accomplish this, you need to create a Python3 program named `sol1.py` that prints a line to be passed as input to the target. Your input needs to overwrite another variable stored on the stack. You do not need to use the shellcode for this target. Test your program with the command:

```
python3 sol1.py | ./vulnerable1
```

HINTS: In Python3, you can write strings containing non-printable ASCII characters by using the escape sequence `\xhh`, where `hh` is a 2-digit hex value (so for example, `\x5c`). To cause Python3 to repeat a character n times, you can do: `sys.stdout.buffer.write(b"\xhh"*n)`. Please ensure the hex is printed as a byte string by prefixing it with `b`. Use `sys.stdout.buffer.write` instead of `print` to avoid issues with ASCII characters and prevent adding a newline character.

4.2 vulnerable2: Overwriting the return address (10 points)

This program takes in a cmdline argument and prints a message:

```
You failed the course.
```

Your task is to provide a cmdline argument that causes the program to print:

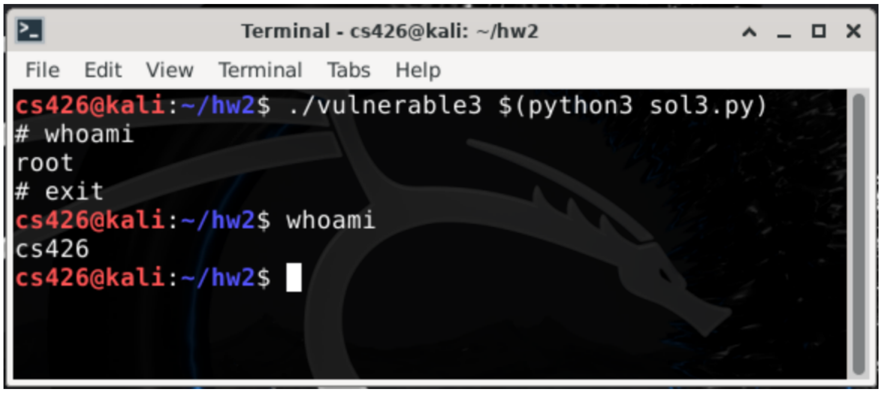
```
You successfully completed the course.
```

To accomplish this, you need to create a Python3 program named `sol2.py` that prints a line to be passed as a cmdline argument to the target. Your command-line argument needs to overwrite the return address so that the function `vulnerable()` transfers control to `print_success()` function when it returns. You do not need to use the shellcode for this target. You can test your program with the command:

```
./vulnerable2 $(python3 sol2.py)
```

4.3 vulnerable3: Redirecting control to shellcode (10 points)

In vulnerables 3 and 4, your goal is to gain root privilege as a normal user by exploiting buffer overflow vulnerabilities in the target programs. These target programs are owned by the `root` user and have the `suid` bit set. Under normal execution, these programs read in a cmdline argument, copy it onto stack, and exit. Your task is to provide a cmdline argument that causes the program to spawn a root shell, as in Figure 1.



Terminal - cs426@kali: ~/hw2
File Edit View Terminal Tabs Help

```
cs426@kali:~/hw2$ ./vulnerable3 $(python3 sol3.py)
# whoami
root
# exit
cs426@kali:~/hw2$ whoami
cs426
cs426@kali:~/hw2$
```

Figure 1: Successful run.

To accomplish this, you need to create a Python3 program named `sol3.py` that prints a line to be passed as a cmdline argument to the target. You need to first place a shellcode (e.g., code to start a shell) inside the program's memory. Then, you need to overwrite the return address so that the implanted shellcode is executed. You can test your program with the command:

```
./vulnerable3 $(python3 sol3.py)
```

On success, you will see a shell prompt (`#`), and running `whoami` will output "`root`" as in Figure 1.

Do not try to create a solution that depends on you manually setting environment variables. You cannot assume that the grader will run your solution with the same environment variables that you have set. Please make sure that your program exits gracefully when you close the spawned shell as well (it should not segfault).

HINTS: You should use the shellcode we have provided in `shellcode.py`[2]. You can simply import it in your solution script (`from shellcode import shellcode`). You are not allowed to use any other (custom) shellcode in any of the exercises. Successfully placing this shellcode in memory and setting the instruction pointer to the beginning of the shellcode (e.g., by returning or jumping to it) will open a shell.

[2] For reference, this shellcode is based on Aleph One's shellcode in [One96].

4.4 vulnerable4: Redirecting control to shellcode through $ebp (10 points)

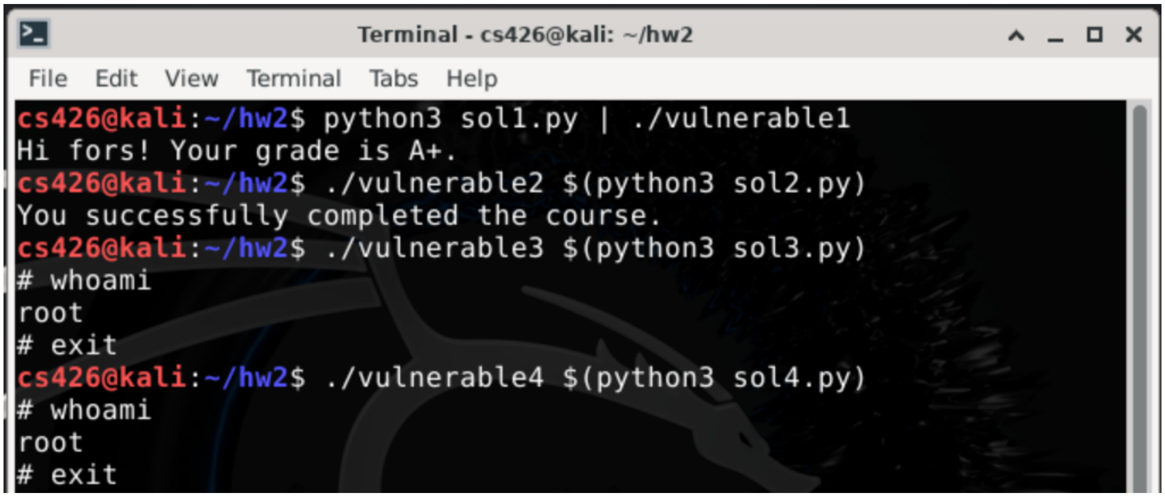
Similar to the last task, your goal is to gain the root privilege as a normal user by exploiting a buffer overflow vulnerability. However, this time you can only overwrite one single byte after the buffer. Create a Python3 program named `sol4.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./vulnerable4 $(python3 sol4.py)
```

On success, you will see a shell prompt (`#`), and running `whoami` will output ”`root`” as in Figure 1.

5 Autograding

We will grade your attack scripts with an autograder (not the one on Gradescope), under the same Kali VM with the same commands as in Section 4. Your attack scripts should **NOT** cause the target programs to print any extra characters or white space, or crash with a segmentation fault. The screenshot below shows the expected outputs for successful attacks.


Terminal - cs426@kali: ~/hw2
File Edit View Terminal Tabs Help

```
cs426@kali:~/hw2$ python3 sol1.py | ./vulnerable1
Hi fors! Your grade is A+.
cs426@kali:~/hw2$ ./vulnerable2 $(python3 sol2.py)
You successfully completed the course.
cs426@kali:~/hw2$ ./vulnerable3 $(python3 sol3.py)
# whoami
root
# exit
cs426@kali:~/hw2$ ./vulnerable4 $(python3 sol4.py)
# whoami
root
# exit
```

6 Deliverable

6.1 What and How to Submit

You need to submit your attack scripts and a report on **Gradescope**, separately.

The PDF report should be submitted on Gradescope under *Homework 2: Basic Buffer Overflow Attacks(Report)*. See the next subsection (Section 6.2) for what to include in the report. Please assign pages to each attack on Gradescope.

The attack scripts should be submitted on Gradescope under *Homework 2: Basic Buffer Overflow Attacks(Code)*. You can use the `prepare_submission.sh` provided to install `zip` and compress all files needed for submission into `submission.zip` as below:

```
cd /home/hw2; chmod +x prepare_submission.sh; ./prepare_submission.sh
```

You will get a warning if there is a file missing. You can update the FILES list inside the script to include other files you want to submit.

We will perform only basic checks on Gradescope (e.g. file naming and cookie value check). After the deadline, we will test your attack scripts using another autograder as specified in Section 5.

File	**Description**	**Submission**
`report.pdf`	Your write-up answering the questions in Section 6.2.	**pdf** on Gradescope [Homework 2 (Report)]
`cookie`	Your cookie file used for personalizing the `vulnerables` (Section 3)	All remaining files should be submitted on Gradescope [Homework 2 (Code)]
`sol1.py`	Your exploit file attacking `vulnerable1` (Section 4.1)	
`sol2.py`	Your exploit file attacking `vulnerable2` (Section 4.2)	
`sol3.py`	Your exploit file attacking `vulnerable3` (Section 4.3)	
`sol4.py`	Your exploit file attacking `vulnerable4` (Section 4.4)	
`other files`	Any additional files or programs created to support your efforts (Section 3.1)	

Table 1: List of required submission files

6.2 Homework Report Instruction

Your submission should include a report/write-up containing enough instructions and technical details to prove the **validity** and **reproducibility** of your work and demonstrate your understanding of each problem and your solution. More specifically, your document should contain detailed answers to the following questions for all `vulnerable`'s:

(a) What does the program do? Describe the behavior of the program.

(b) Why is this program vulnerable (living up to its name)? Identify the vulnerabilities this program has and explain why/how exactly they are vulnerabilities.

(c) What could happen to the machine running this program? Describe the implication and possible outcome of the vulnerabilities identified above.

(d) How can you exploit the identified vulnerabilities? Describe your strategy for attacking this program. Figures like the stacks we drew during lectures are welcomed.

(e) What was your code? Attach your code.[3] Screenshots are also acceptable. Make sure to keep your code as concise as possible (e.g., remove all unnecessary lines), but leave a sufficient amount of comments so it shows that your code matches the strategy you illustrated in the previous part.

If your code is incomplete, still leave what you have gotten so far and describe the direction you were heading and what your plan was to reach the goal.

(f) What proves that your attack was successful? Attach the screenshot(s) of the result of your attack showing that your attack indeed worked and achieved our goal.

If your attack was not successful, still leave a screenshot of the result of your code and describe your debugging experience: What was the failure symptom? Why do you think your code was failing and why do you think it is a reasonable hypothesis? How do you think you could (fix that bug and) make it work?

(g) If you were the author of the program (`vulnerable n`), you would want to 'update' the program so your users do not suffer the same vulnerabilities. How would you remediate/eliminate these vulnerabilities? That is, which line(s) would you replace (and with what)? If this cannot be fixed simply by replacing some lines, what other defense techniques would you use? Please explain with details.

Please be aware: If your report lacks details on many fronts, you may receive a grade close to the grade item for `no attempt or equivalent` on the rubric, EVEN IF your attack is correct. It is also your responsibility to make sure your responses are clean and organized enough to be readable. Similar to Homework 1, meaningless or frivolous responses such as "*because it is correct*", "*trivial*", and "*left as an exercise to the graders*" will be considered completely invalid. In particular, for Part (g), please suggest a fix instead of leaving an unproductive solution such as "*just do not use this program in the first place*" or a copy-paste (or equivalent) of lecture slides.

On the last page of your report, include a **collaboration statement**. It can simply be a list of people you communicated with and the online resources you have used for this homework. If you have finished this assignment entirely on your own AND did not consult any external resources, you MUST indicate that on your collaboration statement. This is a part of your assignment. **Failure to provide one could result in a deduction of points**.

[3] If you are using LaTeX, you may find `lstlisting` environment from the `listings` package useful.

7 Grading and Policy

7.1 Grade Breakdown

Each attack is worth 10 points, with 5 points for the code and 5 points for the corresponding write-up in your report. In total, this homework is worth 4% of your final grade.

As metioned in Section 5, the code will be graded with an autograder. The write-up will be graded based on the rubrics in Section 6.2, with a focus on your understanding of the vulnerabilities, your attack strategies, and your proposed mitigations.

Even if your attacks failed, you can still earn half of the points through your write-up, so don't give up! If your exploit codes keep failing and you are unable to fix them, submit your best attempt along with a detailed explanation in your report. Clearly describe your approach, what you were trying to accomplish, and where you believe your code is failing. Depending on your reasoning and how close you are to the correct solution, you may still receive partial credit.

7.2 Important Rules and Reminders

This assignment MUST be completed using only `gdb` and `python3`, along with necessary general-purpose tools such as `gcc` and `make`. DO NOT use other special-purpose exploit tools such as `ghidra`. No points will be given to a solution that utilizes such tools. For all attacks, DO NOT try to create a solution that depends on you manually setting environment variables.

You are more than welcome to use any external resources including the ones available online besides the ones provided to you already (e.g., [One96]). However, the academic integrity policy still holds. **Do not ask someone else to do your work for you** or copy and paste the questions on websites to find answers. **Use your resources reasonably, and do not cross the line**. See the **Academic Integrity** section of the syllabus (course webpage) for more details.

References

[One96] Aleph One, *Smashing the stack for fun and profit*, 1996, Available Online: `https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf`.