CS 426: Computer Security
Fall 2025
Homework 4: Memory Safety Attacks

Due: Sep 25th, 2025 @ 11:59 PM

1 Overview

The goal of this assignment is to help you gain hands-on experience on return-oriented programming and heap overflow attacks.

2 Environment Setup

Redownload an updated Kali image from https://app.box.com/s/igf1ky7ert6tm8r0zm21bx77e0yyjzkx**. You must use the new 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 provided Kali Linux VM 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.

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*.

3 Assignment Setup

Files for the assignment are already included in the new VM. Run the VM, open the terminal and navigate to `/home/cs426/hw4`. 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 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`. The only exceptions are the tools such as `ROPgadget` and `Ropper` which can be used for attacking `vulnerable1` (See Section 4.1).

GDB: You will find the GDB debugger extremely useful for finding buffer and heap overflow exploits. Make use of your prior experience from Homework 1 and 2, and do not be afraid to experiment! This quick reference may also be useful: https://web.eecs.umich.edu/~sugih/pointers/Gdb-reference-card.pdf.

x86 Assembly: There are many good references for x86 assembly language but note that this problem 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, not x86_64.

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 Stack and Heap Overflow Targets

This section consists of four targets/vulnerables. Make sure that you read the submission instructions in Section 6. The target programs are simple, short C programs with (mostly) clear security vulnerabilities. We provided the source codes and the Makefile. You will use the Makefile to compile all the targets into their binaries as explained in Section 3.

Your exploits must work against the targets as compiled and executed within the provided VM. For all programs, please use the version of Python3 that is the default on the VM (i.e., just call `python3` as specified below, do not change the links, use a different version, etc.).

Before starting the steps below, make sure that you generated your `cookie` using the `setcookie` executable with your username. **Otherwise, we cannot grade your work.**

4.1 vulnerable1: Return-oriented programming (10 points)

This simple program has a clear buffer overflow vulnerability that would allow you redirect the execution to shellcode by overwriting the return address. However, **you are not allowed to use shellcode** this time. Implement an ROP-based attack to bypass DEP and open a `root` shell. Using tools such as Ropper and ROPgadget is highly recommended and makes your life easier on this attack. These tools are already installed in the provided VM.

1. Though there are a number of ways you could implement a return-oriented program, your ROP should use the `execve` system-call to run the `"/bin/sh"` binary. This is equivalent to:

   ```
   execve("/bin/sh", 0, 0)  or  execve("/bin/sh", "", "")
   ```

2. For an extra push in the right direction, `int 0x80` is the assembly instruction for interrupting execution with a syscall, and if the EAX register contains the number 11, it will be an `execve`. Now all you need to figure out is what values you need for the inputs of that function which need to be in EBX, ECX, and EDX, and how to set them using ROP gadgets! Some gadget addresses may contain some ASCII characters (e.g., `\x0a` for a newline) that could be ignored or break your payload. If so, try a different combination of gadgets to achieve the same effect.

3. Even though tools like Ropper and ROPgadget are very useful for finding and creating ROP chain, as we saw in class, they are not perfect. For this assignment and real world attacks, the chain generated by these tools might not work out of the box and you may need to use a direct search for some gadgets you need (see the HINT below).

4. **HINT:** Since some bytes cannot be used in your payload, there will be a limited number of gadgets for setting some registers, more specifically ECX register. You can use a gadget like `xchg ecx, eax;` for that purpose.

To accomplish the task, you need to create a Python3 program named `sol1.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./vulnerable1 $(python3 sol1.py)
```

You may find the already installed `objdump` utility helpful. In addition, the `.data` segment of the program can be useful to store strings and null bytes since it stores the global and static variables.

For this target, **it is acceptable if the program segfaults after the root shell is closed**.

4.2 vulnerable2: Beyond strings (10 points)

This target program takes as its command-line argument the name of a data file it will read. The file format is a 32-bit count followed by that many 32-bit integers. Create a data file that causes the provided shellcode to execute and opens a root shell.

Create a Python3 program named `sol2.py` that outputs the contents of a data file to be read by the target. Test your program with the command line:

```
python3 sol2.py > tmp; ./vulnerable2 tmp
```

4.3 vulnerable3: Heap-based exploitation (10 points)

This program implements a doubly linked list on the heap. It takes three command-line arguments. Figure out a way to exploit it to open a root shell using the shellcode.

To accomplish this, you need to create a Python3 program named `sol3.py` that prints lines to be used for each of the command-line arguments to the target. Your program should take a single numeric argument that determines which of the three arguments it outputs. Test your program with the command line:

```
./vulnerable3 $(python3 sol3.py 1) $(python3 sol3.py 2) $(python3 sol3.py 3)
```

For this target, **it is acceptable if the program segfaults after the root shell is closed**.

4.4 vulnerable4: Heap unlink exploitation (Bonus) (2 points)

This challenge requires a good understanding of the glibc heap implementation. Completing this task is **not mandatory**, but you will receive **extra credit** if you complete. You should attempt it **only after you have completed the other challenges**.

This program allocates two buffers on the heap, and populates them by reading from the files provided as arguments. The command-line arguments should be the name of the files that contain your payload. Figure out a way to make the program output "You have the correct access rights!".

To accomplish this, you need to create a Python3 program named `sol4.py` that prints lines to be saved in payload files. Then you should pass the filenames as each of the command-line arguments to the target. Your program should take a single numeric argument that determines which of the three files it outputs. Test your program with the command line:

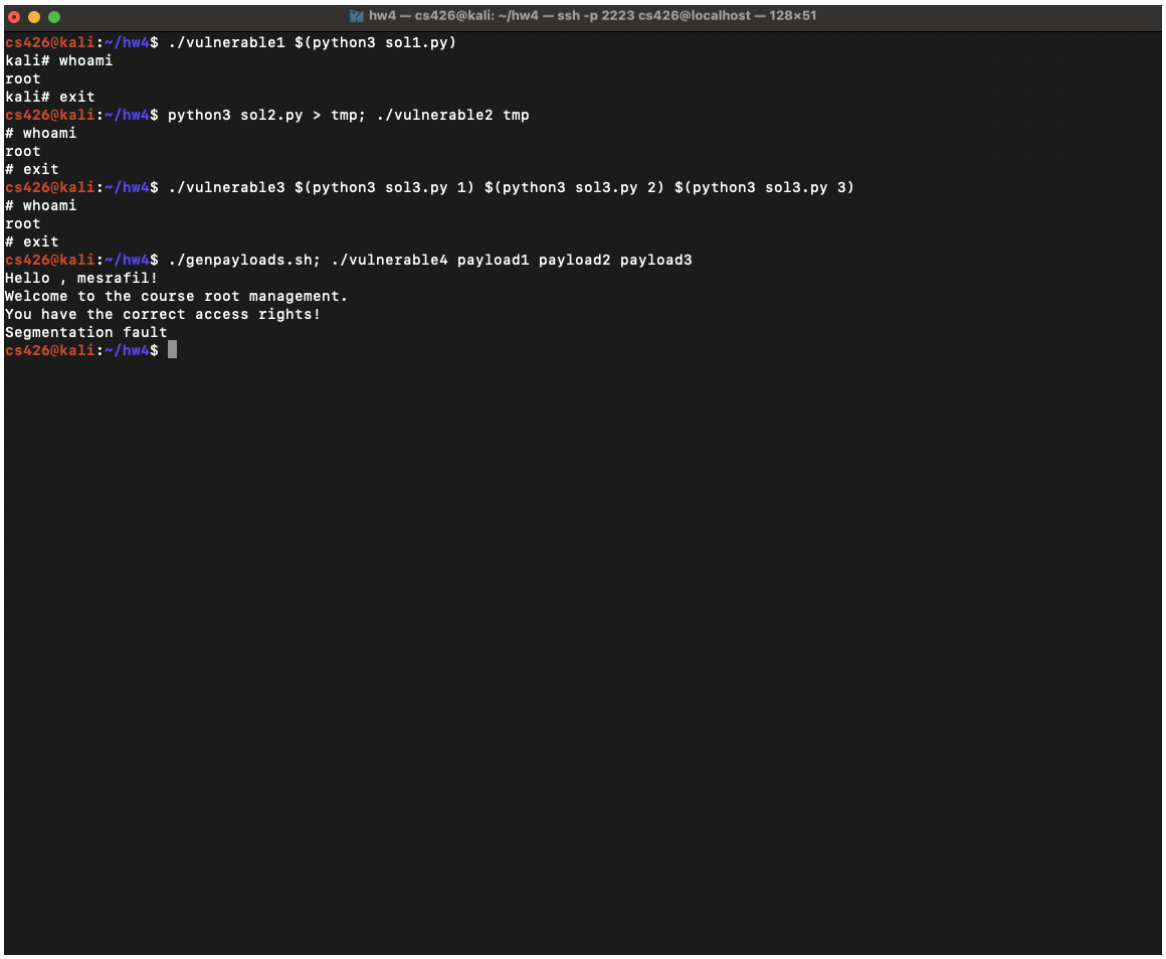
```
./genpayloads.sh; ./vulnerable4 payload1 payload2 payload3
```

You will need to apply `./genpayloads.sh` every time you update your attack code. For this target, **it is acceptable if the program segfaults after execution**.

We recommend checking out unlink exploit explanation and malloc chunk structure. These documents will significantly help you solve this challenge.

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.



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 4: Memory Safety 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 4: Memory Safety 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/hw4; 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.

File	Description	Submission
`report.pdf`	Your write-up answering the questions in Section 6.2.	**pdf** on Gradescope [Homework 4: Memory Safety Attacks (Report)]
`cookie`	Your cookie file used for personalizing the `vulnerables` (Section 3)	All remaining files should be submitted on Gradescope. [Homework 4: Memory Safety Attacks (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 (bonus)`	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 the **vulnerable**s:

(a) Why is this program vulnerable (living up to its name)? Identify the vulnerabilities this program has and explain why/how exactly they are vulnerabilities. [**2-3 sentences**]

(b) How can you exploit the identified vulnerabilities? Describe your strategy for attacking this program. Figures like the stacks we drew during lectures are welcomed. [**2-3 sentences**]

(c) What was your code? Attach your code.[1] 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. [**No explanations on the writeup needed**]

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. [**As detailed as possible**]

(d) 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. [**No explanations on the writeup needed**]

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? [**As detailed as possible**]

(e) If you were the author of the program (`vulnerable n` or `sysapp.c`), 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? Note that for sysapp.c, simply removing the delay is not an acceptable solution here. [**2-3 sentences**]

Please be aware: It is your responsibility to make sure your responses are clean and organized enough to be readable. Similar to Homework 1 and 2, 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 (e), please suggest a fix instead of leaving an unproductive solution such as "*just don't 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**.

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

7 Grading and Policy

7.1 Grade Breakdown

There are three mandatory attacks and a bonus attack in this assignment. Each mandatory 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. The bonus attack is worth 2 points, with 1 point for the code and 1 point for the write-up.

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. The only exceptions are the tools such as `ROPgadget` and `Ropper` which can be used for attacking `vulnerable1` (See Section 4.1). 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. 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.