CS 426: Computer Security
Fall 2025
Homework 5: Side-Channel Attacks

Due Oct. 16th, 2025 @ 11:59 PM

1 Overview

The goal of this assignment is to help you gain hands-on experience with side-channel attacks. To better understand these attacks, you can review the lecture on September 30 and October 2. You may also find many helpful resources online, such as this short video that introduces timing side-channel attacks in an interesting way.

2 Environment Setup

Use the provided ARM server for this assignment. Side-channel attacks, especially timing-based attacks, are highly sensitive to the exact hardware and software environment. For this reason, you must use the provided ARM server (`armor0.cs.purdue.edu`) to develop and test your side-channel experiments. We will grade submissions using the same ARM server. Therefore, do not attempt to run targets on other machines. Small differences in CPU microarchitecture, frequency scaling, etc. can make an attack succeed or fail; using a personal laptop or a different server will produce results that are not comparable to the canonical grading environment.

To connect to the ARM server, you need to be on Purdue networks, either by connecting to PAL network, using Purdue VPN, or using `data.cs.purdue.edu` as a proxy. Afterwards, you can `ssh` into the ARM server by running the following command, using your Purdue Career username(e.g., mesrafil).

```
ssh <username>@armor0.cs.purdue.edu   # replace <username> with your Purdue username
```

To view the password for your account, visit Gradescope and view it under **ARM Server Passwords**.

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

1. On the ARM server, copy the `hw6` starter code from the course shared directoy.

```
ssh <username>@armor0.cs.purdue.edu    # ssh into the server
cp -r /home/cs426/hw6 ~/               # copy hw6 to your home directory
cd hw6
```

2. Generate the randomized target code. First, set a cookie using your Purdue Career username (e.g., `./setcookie mesrafil`). Then, generate the randomized target source codes based on the cookie:

```
./setcookie <username>           # replace <username> with your Purdue username
make generate
```

3. Compile the randomized target together with the attack starter codes. The starter codes (*memhack.c* and *timehack.c*) only print the error message *"Could not get the password!"*. Your task in this assignment is to complete the attacker codes so they perform the required attacks.

```
make                      # recompile every time you modify the attack codes
./memhack                 # gives error "Could not get the password!"
./timehack                # gives error "Could not get the password!"
```

Now you can start attacking the target binaries! Note that you are required to **not use 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`.

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 Side-Channel Attacks

This section consists of one target, a simple C program with (mostly) clear security vulnerabilities. Your task is to use two different side-channel attacks to extract a secret from the target. **The secret can be any length, but no longer than 16 characters. It consists only of letters (A-Z, a-z) and numbers (0-9), excluding all other characters.**

Do not write a solution that simply checks all passwords exhaustively. You will not get credit for this. Both attacks should be exploitable in linear time (we will stop programs that are running for excessive periods) and it will basically feel instantaneous for passwords of 12 characters.

Make sure that you read the submission instructions in Section 6. 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. Make sure that you generated your `cookie` using the `setcookie` executable with your username. **Otherwise, we cannot grade your work.**

4.1 Memory-Based Side-Channel Attack (10 points)

We recommend starting with the memory-based side channel attack because it is deterministic and less affected by noise. Begin by examining the `check_pass()` function in `sysapp.c` and take note of two key details:

1. The password string is passed by reference.
2. The memory it points to is checked against the reference password one character at a time.

Next, look at `memhack.c` and observe how a buffer is allocated. The memory region starting at `page_start` is protected, meaning accessing it triggers a segmentation fault (SEGV), while the previous 32 characters are allocated for use. Also, review the `demonstrate_signals()` function, which shows how accessing memory in the protected page causes a fault and how to catch and handle that fault in your program. You do not need to use `demonstrate_signals()` directly, but it demonstrates how to use signals to determine whether a memory reference touches a protected page.

Your goal is to create an attack in `memhack.c` that calls `check_pass()` with different inputs, catches resulting faults, and uses those faults to determine correct password characters. To do this, iterate through the maximum password length of 16 characters. For each guess, store the password so that the first character is positioned one byte before `page_start`, then iterate through possible character choices. When the correct character is found, a page fault will occur. Repeat this process for all 16 password characters. The password may contain any letters (A-Z, a-z) and numbers (0-9), excluding all other characters.

Other hints:

1. You already have a page-protected buffer with enough space to crack the password, so use it appropriately for each guess.
2. The `demonstrate_signals()` function already handles segmentation faults for you, and most of it can be reused in your code.

4.2 Timing-Based Side-Channel Attack (10 points)

We provide the skeleton code `timehack.c` that you will implement to exploit the timing-based side-channel vulnerability present in the target library `sysapp.c` to obtain a protected key. Note that you should not modify the library code `sysapp.c`, but only `timehack.c`. Once your exploit can determine the password in `sysapp.c` and call the `hack_system()` function successfully in a reasonable amount of time, your attack is successful. Again, make sure that you read the submission instructions in Section 6.

In this timing-based side-channel attack, you will deal with noise. Please go and take a look at `check_pass()` in `sysapp.c`. An artificial delay has been added when each character is checked to make your life easier (it is possible to do the assignment without it but it would require a much more careful methodology). The execution time of `check_pass()` depends on how many characters you guess correctly.

Look in `timehack.c` and find a macro there for a function called `read_cntvct()` which invokes the processor's cycle counter (a clock that increments by one for each processor cycle that passes). In general, treat `read_cntvct()` as a free running timer that returns a timestamp. Insert a call to `read_cntvct()` before the call to `check_pass()` and afterwards. Print the difference between these values to see how long (in cycles) the password checking ran. Run the program a few times. Now change the guess string so the first character is correct. Run again and see how the time difference changes.

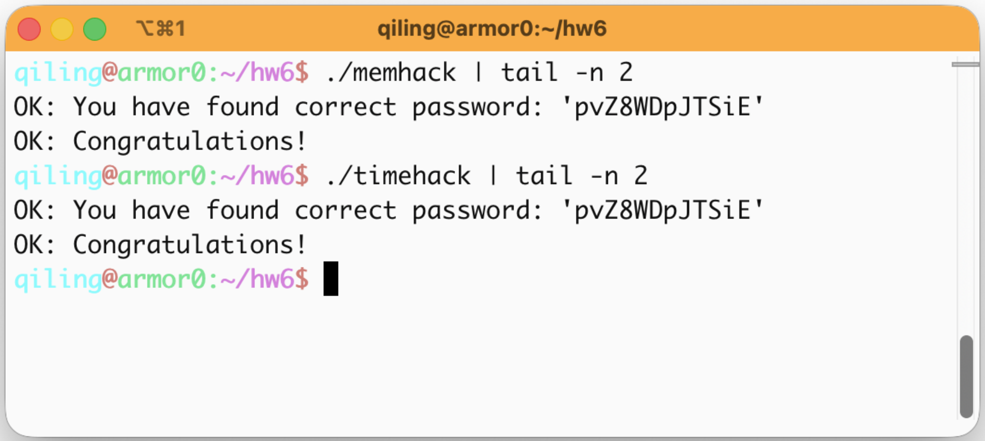
Now automate this entire process. Even though autograding is performed on the same machine, absolute execution times can still fluctuate due to transient system activity, CPU scheduling, cache warm-up effects, and other background noise. Therefore, **focus on the relative timing differences** between your measurements rather than the raw timing values. Running multiple trials per character and averaging the results will help reduce noise and produce more consistent conclusions.

Other hints:

1. Be careful in using `printf`'s. These can blow out the instruction cache and data caches and perturb your results (i.e., overwhelm the timing effects you are trying to detect).
2. Be careful in averaging across trials. If your process is descheduled in the middle of a measurement, the time cost of that individual trial will be so large that it overwhelms everything else. Instead, the **median** is your friend. However, feel free to experiment if something does not work for you.
3. If time is not continuing to increase as you progress through characters, then you probably made a bad guess earlier. Backtrack.
4. Debugging advice: Make a big array to hold your timing measurements and print them at the end.
5. Be sure to test a bunch of different passwords. We will also test with different passwords when we are grading your work.

5 Autograding

We will grade your attack scripts with an autograder (not the one on Gradescope), under the same ARM server. For both attacks, your attack codes should call the *hack_system* function with the password you hacked. The screenshot below shows the expected outputs for successful attacks.



```
qiling@armor0:~/hw6$ ./memhack | tail -n 2
OK: You have found correct password: 'pvZ8WDpJTSiE'
OK: Congratulations!
qiling@armor0:~/hw6$ ./timehack | tail -n 2
OK: You have found correct password: 'pvZ8WDpJTSiE'
OK: Congratulations!
qiling@armor0:~/hw6$
```

Note that it's fine if your attack codes print some debug information. However, don't print anything after the final call to the *hack_system*.

6 Deliverable

6.1 What and How to Submit

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

The PDF report should be submitted on Gradescope under *Homework 6: Side-Channel 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 6: Side-Channel Attacks(Code)*. You can use the `prepare_submission.sh` provided to compress all files needed for submission into `submission.zip` as below:

```
cd ~/hw6; 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 codes 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 6 (Report)]
`cookie`	Your cookie file used for personalizing the targets (Section 3)	All remaining files should be submitted on Gradescope [Homework 6 (Code)]
`memhack.c`	Your exploit file attacking `sysapp.c` library (Section 4.1)	
`timehack.c`	Your exploit file attacking `sysapp.c` library (Section 4.2)	
`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 both attacks:

(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.[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.

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 (`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? 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**.

[1] 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`, along with necessary general-purpose tools such as `gcc`, `make`, and `python`. 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. 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.