OCR@UC Lab

LESSON TITLE: **Lab – Meltdown & Spectre**

WARNING:

Warning: Any use of penetration testing techniques on a live network could result in expulsion and/or criminal prosecution. Techniques are to be used in lab environments, for educational use only or on networks for which you have explicit permission to test its defenses.

Level:

☐Beginner

☒Intermediate

☐Advanced

Time Required: 90 minutes

Audience: ☒Instructor-led

☐Self-taught

Lesson Learning Outcomes: Upon completion of this lesson, students will be able to:

Demonstrate the process of how Spectre and meltdown attack can be performed.

Materials List:

- Computers with Internet connection
- Browsers: Firefox (preferred), Google Chrome, or Internet Explorer
- Intro to Ethical Hacking lab environment

Introduction

System and Tool Used:

- Kali Linux *(u: root, p: toor)*
  - C, knowledge of memory mapping, kernel module(How to load kernel module(for meltdown attack))

- Power down all other systems

Module Activity Description: **Time Required: ~30 minutes**

Part One: **Spectre attack**

What is a spectre attack??

Spectre attacks induce a victim to speculatively perform operations that would not occur during strictly serialized inorder processing of the program's instructions, and which leak victim's confidential information via a covert channel to the adversary.

Spectre and meltdown attack are using some properties of processors which are present mostly in the latest processors. Some of these properties are explained below:

A. Out-of-order Execution

An out-of-order execution paradigm increases the utilization of the processor's components by allowing instructions further down the instruction stream of a program to be executed in parallel with, and sometimes before, preceding instructions.
Modern processors internally work with micro-ops, emulating the instruction set of the architecture, i.e., instructions are decoded into micro-ops [15]. Once all of the microops corresponding to an instruction, as well as all preceding instructions, have been completed, the instructions can be retired, committing in their changes to registers and other architectural state and freeing the reorder buffer space. As a result, instructions are retired in program execution order.

B. Speculative Execution

Often, the processor does not know the future instruction stream of a program. For example, this occurs when out-oforder execution reaches a conditional branch instruction whose direction depends on preceding instructions whose execution is not completed yet. In such cases, the processor can preserve its current register state, make a prediction as to the path that the program will follow, and speculatively execute instructions along the path. If the prediction turns out to be correct, the results of the speculative execution are committed (i.e., saved), yielding a performance advantage over idling during the wait. Otherwise, when the processor determines that it followed the wrong path, it abandons the work it performed speculatively by reverting its register state and resuming along the correct path. We refer to instructions which are performed erroneously (i.e., as the result of a misprediction), but may leave microarchitectural traces, as transient instructions. Although the speculative execution maintains the architectural state of the program as if execution followed the correct path, microarchitectural elements may be in a different (but valid) state than before the transient execution.

C. Branch prediction

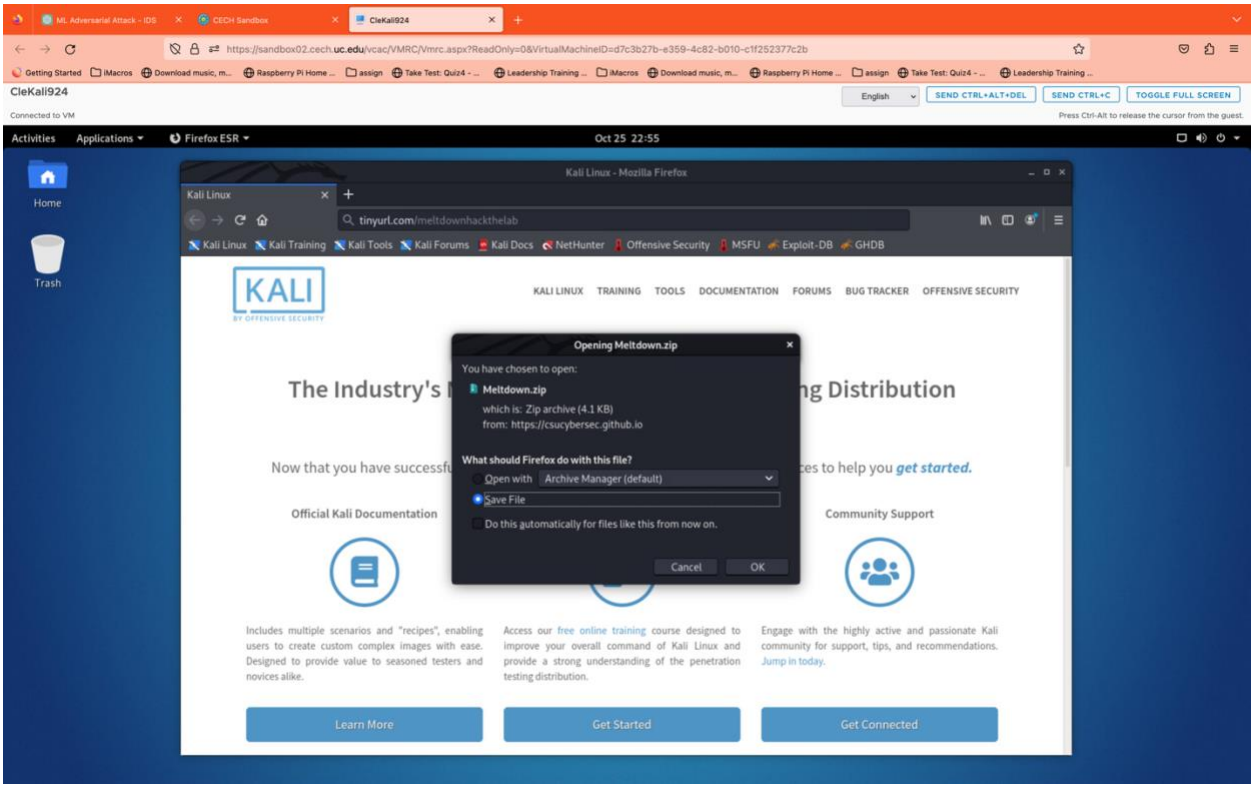
During speculative execution, the processor makes guesses as to the likely outcome of branch instructions. Better predictions improve performance by increasing the number of speculatively executed operations that can be successfully committed. The branch predictors of modern Intel processors, e.g., Haswell Xeon processors, have multiple prediction mechanisms for direct and indirect branches. Indirect branch instructions can jump to arbitrary target addresses computed at runtime. For example, x86 instructions can

jump to an address in a register, memory location, or on the stack e.g., “jmp eax”, “jmp [eax]”, and “ret”. Indirect branches are also supported on ARM (e.g., “MOV pc, r14”), MIPS (e.g., “jr $ra”), RISC-V (e.g., “jalr x0,x1,0”), and other processors. To compensate for the additional flexibility as compared to direct branches, indirect jumps and calls are optimized using at least two different prediction mechanisms . Intel describes that the processor predicts • “Direct Calls and Jumps” in a static or monotonic manner, • “Indirect Calls and Jumps” either in a monotonic manner, or in a varying manner, which depends on recent program behavior, and for • “Conditional Branches” the branch target and whether the branch will be taken. Consequently, several processor components are used for predicting the outcome of branches. The Branch Target Buffer (BTB) keeps a mapping from addresses of recently executed branch instructions to destination addresses. Processors can use the BTB to predict future code addresses even before decoding the branch instructions. Evtyushkin et al. analyzed the BTB of an Intel Haswell processor and concluded that only the 31 least significant bits of the branch address are used to index the BTB.

Above are the properties of some advanced processors based on which spectre and meltdown works. In the spectre attack speculative execution property of a processor is used. Processor executes those instructions which dependent on a condition to make the execution faster and if that conditions are not met, then the results instructions executed remains unattended. These results can be any secrete information like passwords, card details etc.

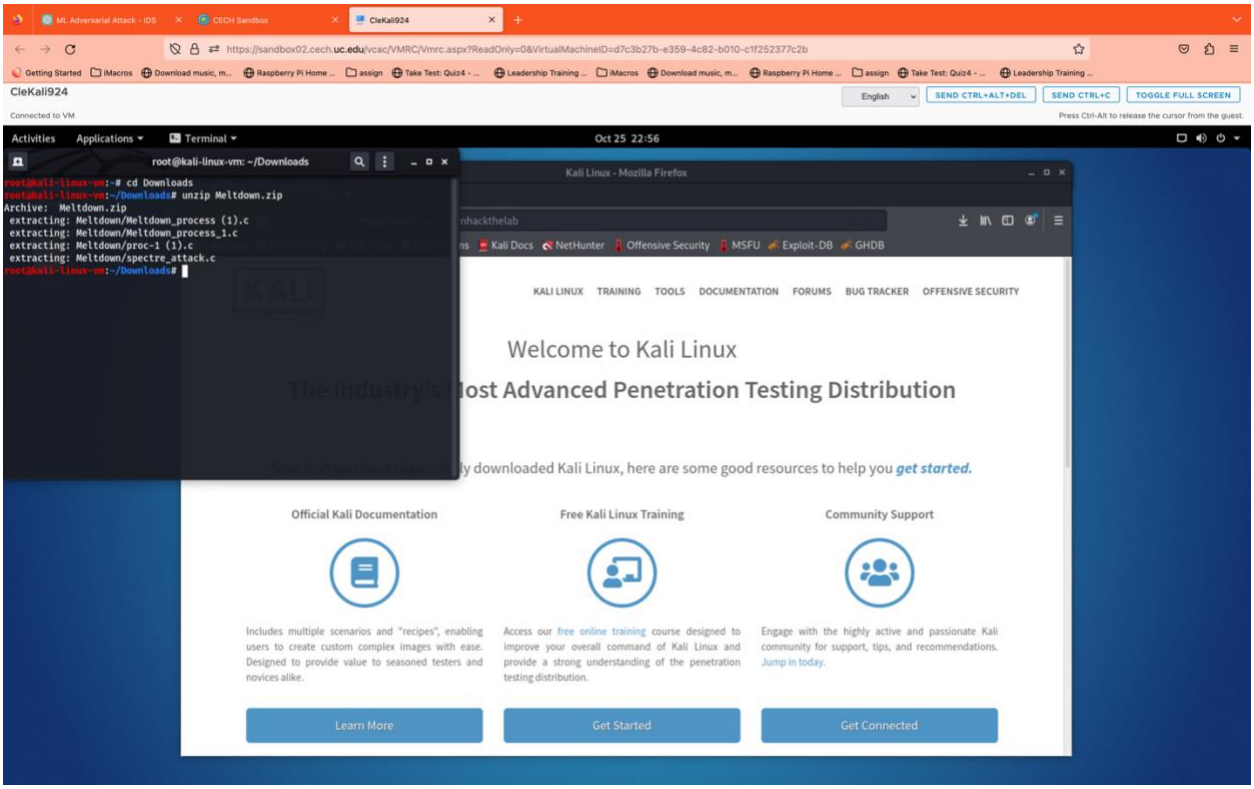
We have written a code below to demonstrate the spectre attack. We will also discuss the output of the code. Write this code in kali linux on OCRI and save it as “.c” file extension.

Download meltdown&spectre codes from this link:
https://tinyurl.com/meltdownhackthelab

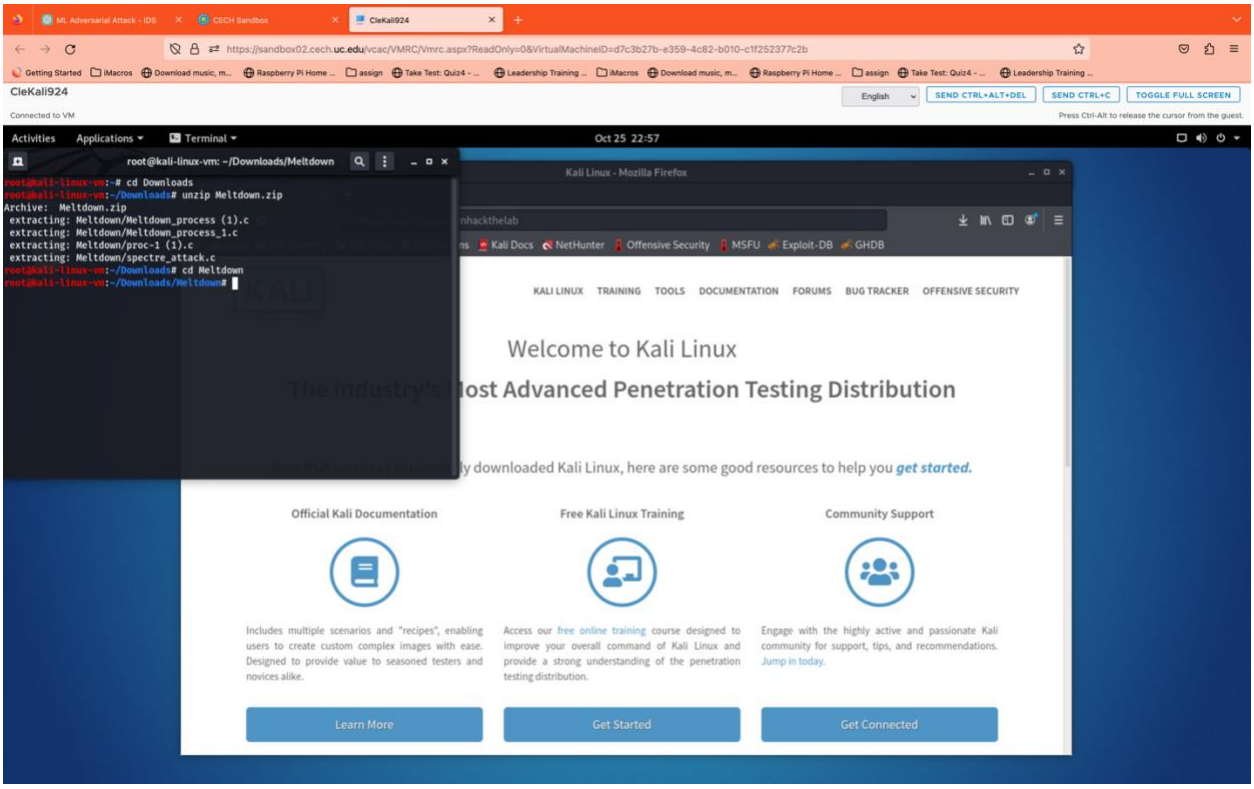


Follow the following commands in terminal.

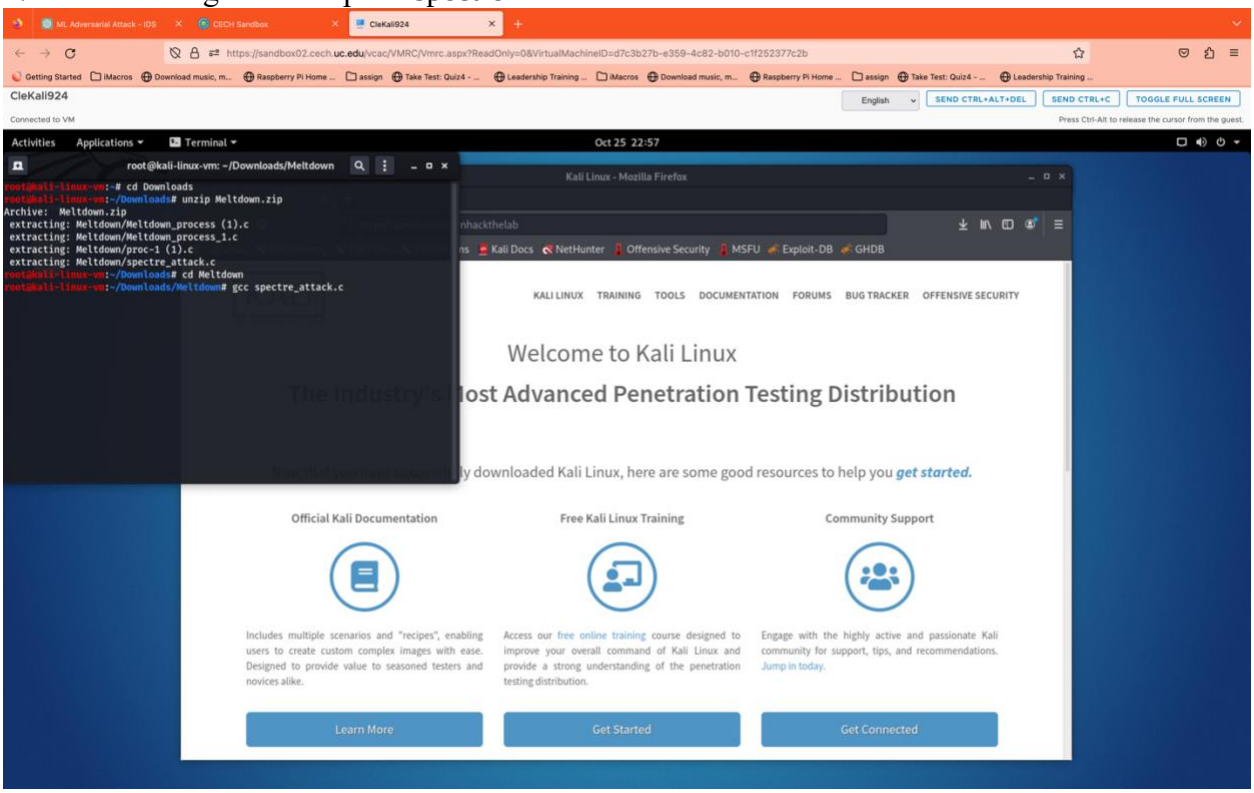
1. cd Downloads
2. unzip Meltdown.zip



3. Cd Meltdown



4. Further use gcc to compile “spectre.c” file and “./a.out” to execute.




```
#include<emmintrin.h>
#include<x86intrin.h>
int size = 10;
```

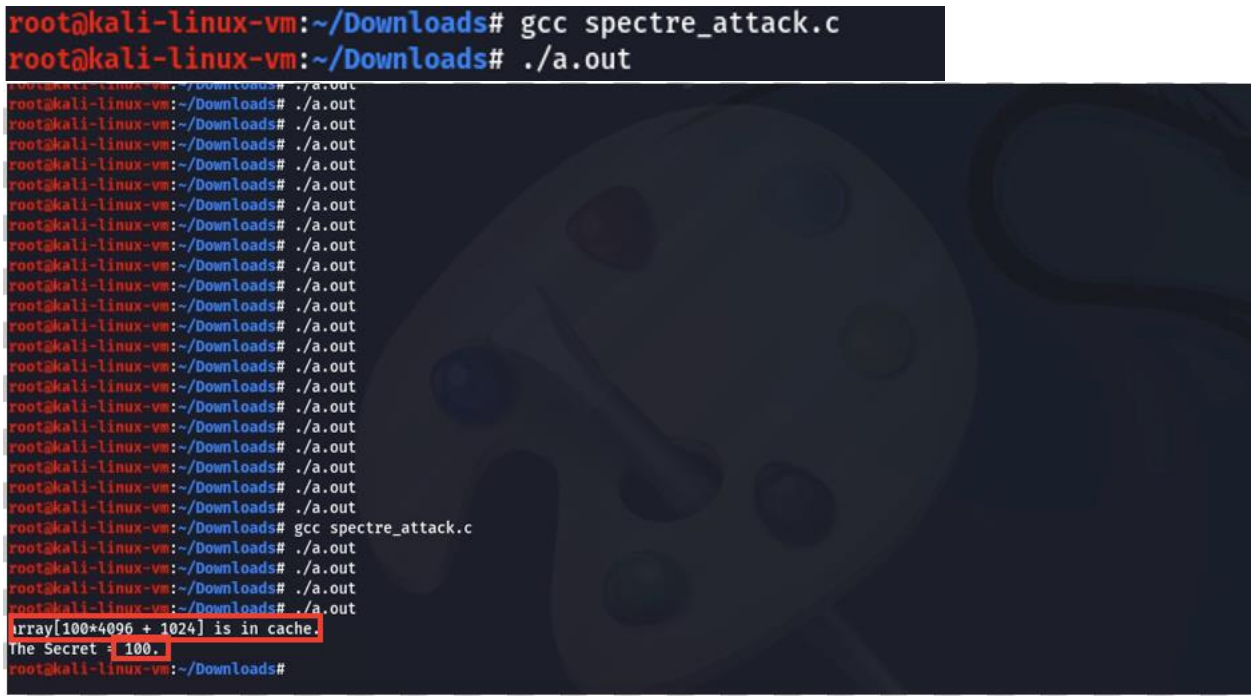
```
 uint8_t array[256*4096];
 uint8_t temp = 0;
#define CACHE_HIT_THRESHOLD (80)
 #define DELTA 1024
void victim(size_t x)
 {
 if (x < size)
 {
 temp = array[x * 4096 + DELTA];
} }

void reloadSideChannel()
{
int junk=0;
register uint64_t time1, time2;
 volatile uint8_t *addr;
 int i;
 for(i = 0; i < 256; i++)
{
 addr = &array[i*4096 + DELTA];
 time1 = __rdtscp(&junk);
 junk = *addr;
time2 = __rdtscp(&junk) - time1;
 if (time2 <= CACHE_HIT_THRESHOLD)
{
printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);
printf("The Secret = %d.\n",i); }
} }


int main()
 {
int i;
flushSideChannel();
 for (i = 0; i < 10; i++)
{
mm_clflush(&size);
 victim(i);
 }
mm_clflush(&size);
for (i = 0; i < 256; i++)
_mm_clflush(&array[i*4096 + DELTA]);
victim(100);
reloadSideChannel();
return (0);
}
```

Here we are using the speculative execution property of a CPU.

To perform speculative execution, CPU should be able to predict the outcome of the if condition. CPUs keep a record of the branches taken in the past, and then use these past results to predict what branch should be taken in a speculative execution.
Hence here we are training the cpu , so that selected branch can become the prediction result.
Inside the loop, we invoke victim() with a small argument (from 0 to 9). These values are less than the value size, so the true-branch of the if-condition is always taken. This is the training phase, which essentially trains the CPU to expect the if-condition to come out to be true. Once the CPU is trained, we pass a larger value (100) to the victim() function. This value is larger than size, so the false-branch of the if-condition inside victim() will be taken in the actual execution, not the true-branch. However, we have flushed the variable size from the memory, so getting its value from the memory may take a while. This is when the CPU will make a prediction, and start speculative execution.
We have to run the code multiple times to get the expected output.



As we can see in the above screen shot we have to run the a.out file made by compiling the spectre_attack.c multiple times.

Question 1: Take a screenshot of the resultant secret value like the following.

The above code provided is a basic example of a cache side-channel attack called a Flush+Reload attack. This attack targets the cache memory of processors and is used to infer the value of a specific data that a victim process accesses.

Module Activity Description: **Time Required: ~30 minutes**

Part Two: **Meltdown-Type 2 attack**

We can perform meltdown attack by reading the memory of a process from another process and changing the content of any particular memory location. We can do that using the /proc/$pid/mem directory. Here $pid is the process of victim process. We can use the code in meltdown_process_1.c to read the memory of another process and change the content of that particular memory location it.

We can also use meltdown_process.c but there are some draw backs in it.

We can open the /proc/$pid/mem file, we can't actually read or write on that file without attaching to the process as a debugger. well'll just get EIO errors. To attach, use ptrace() with PTRACE_ATTACH. This asynchronously delivers a SIGSTOP signal to the target, which has to be waited on with waitpid().

You could select the target address with lseek(), but it's cleaner and more efficient just to do it all in one system call with pread() and pwrite().

To perform a meltdown attack we need run the proc-1(victim process) first.

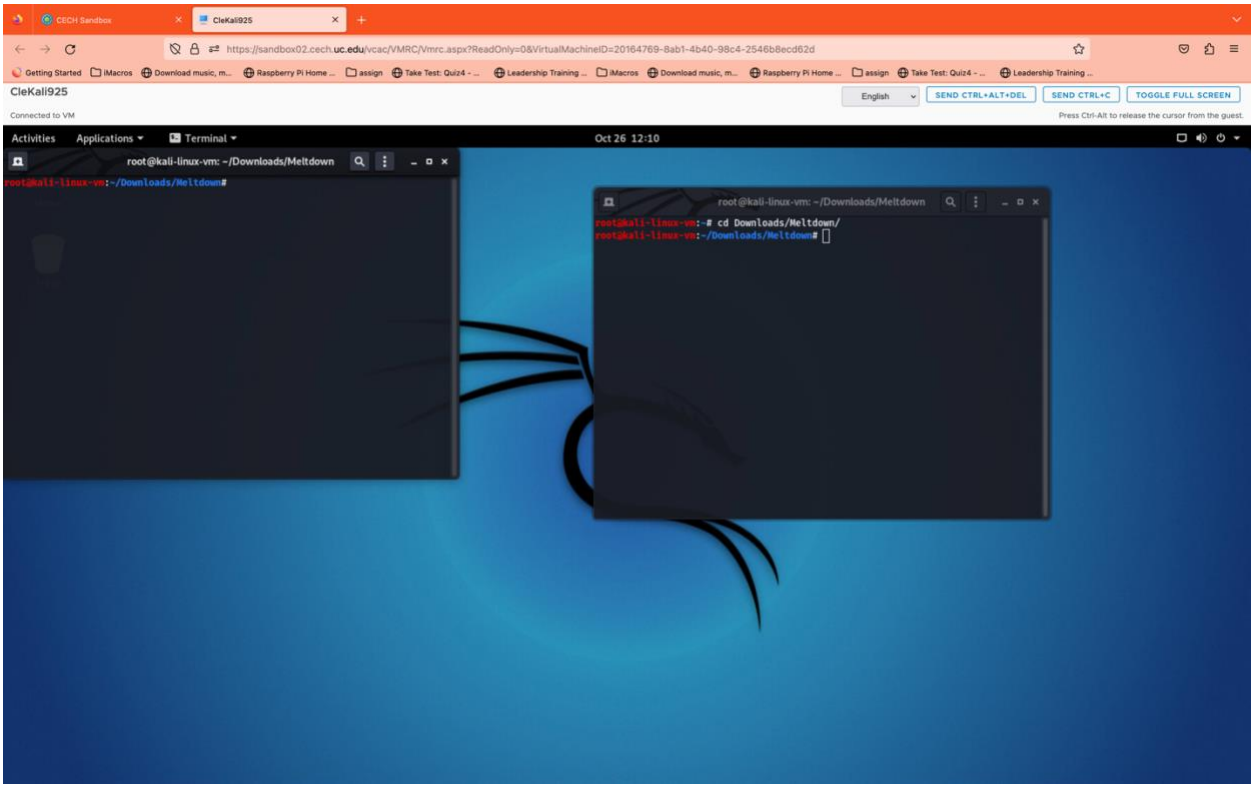
When we run the proc-1 we are printing the pid, address of string we want to read and the length of string we want to read/replace.(we are printing these things in proc-1 for demo purpose , to perform real life attack we can analyze the logs of victim process to get the address of any import tant variables in the process. Pid we can fetch from the ps -a command).

We need to copy the pid, address of string we want to read and the length of string we want to read/replace from the proc-1 and give it to meltdown_process as command line arguments.

Now open 2 terminal and follow the following steps

Navigate both terminal to Meltdown by executing following command.

cd Downloads/Meltdown/



Now on one tab execute following commands

- **gcc proc-1\ \(1\).c**
- **./a.out (Do not execute this command till you complete gcc step of meltdown)**

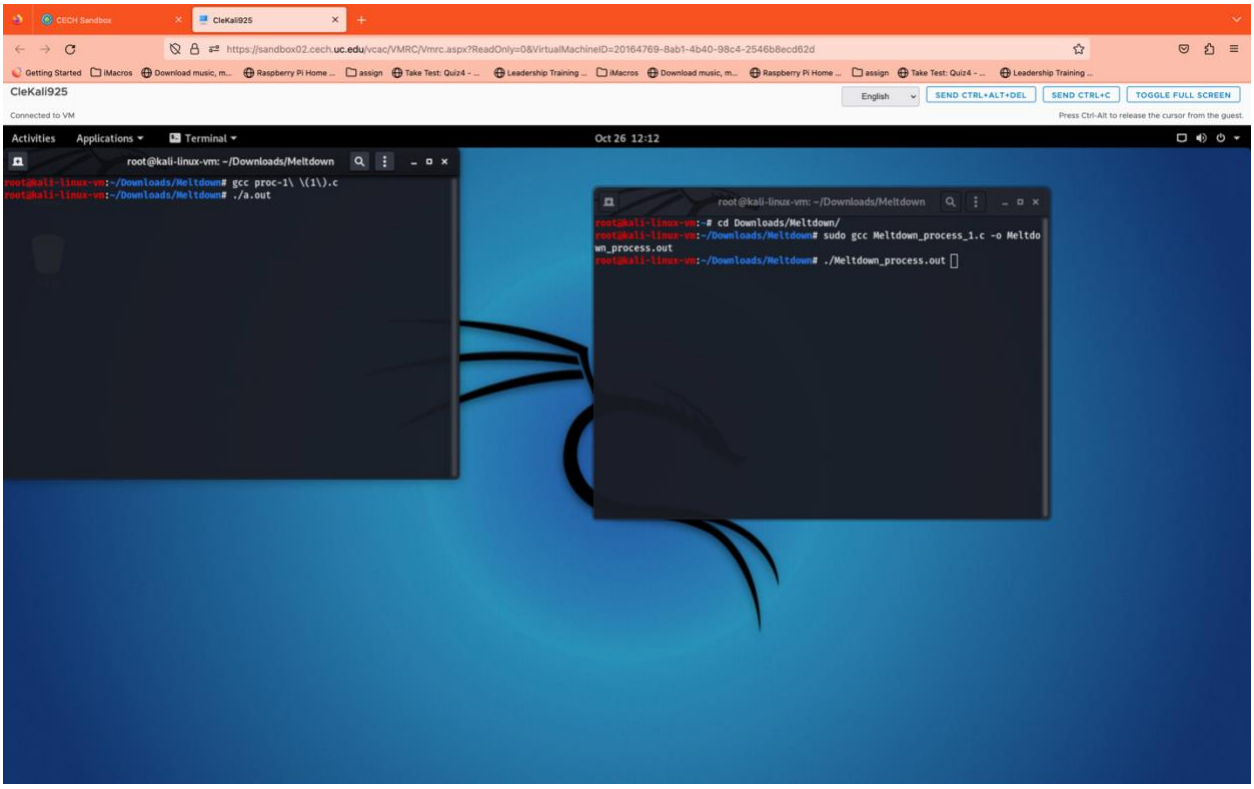
Now perform the following procedure on it.

```
root@kali-linux-vm:~/Downloads/Meltdown# gcc proc-1\ \(1\).c
root@kali-linux-vm:~/Downloads/Meltdown#
root@kali-linux-vm:~/Downloads/Meltdown# ./a.out
 2345  7fffb762d5a0  30
```

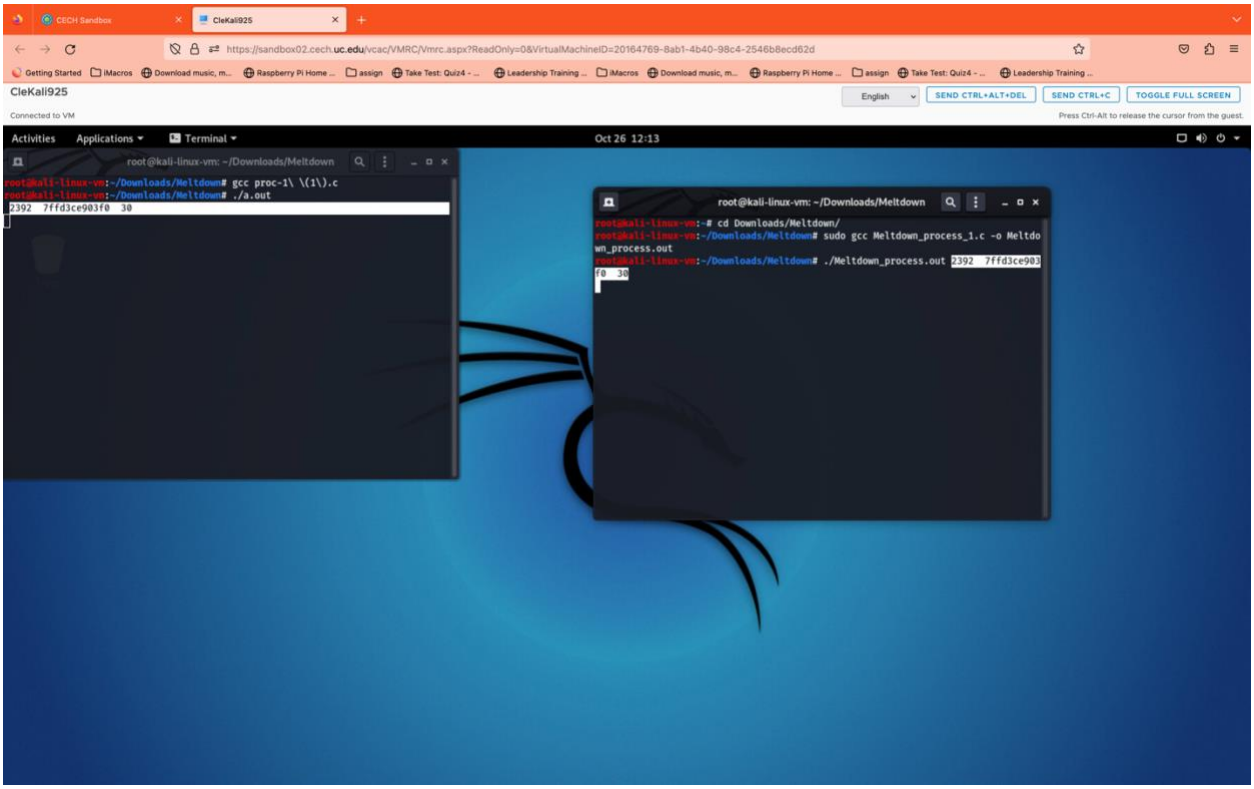
We need to copy data printed in the above screen shot and give it as a input to meltdown process in another tab of terminal.

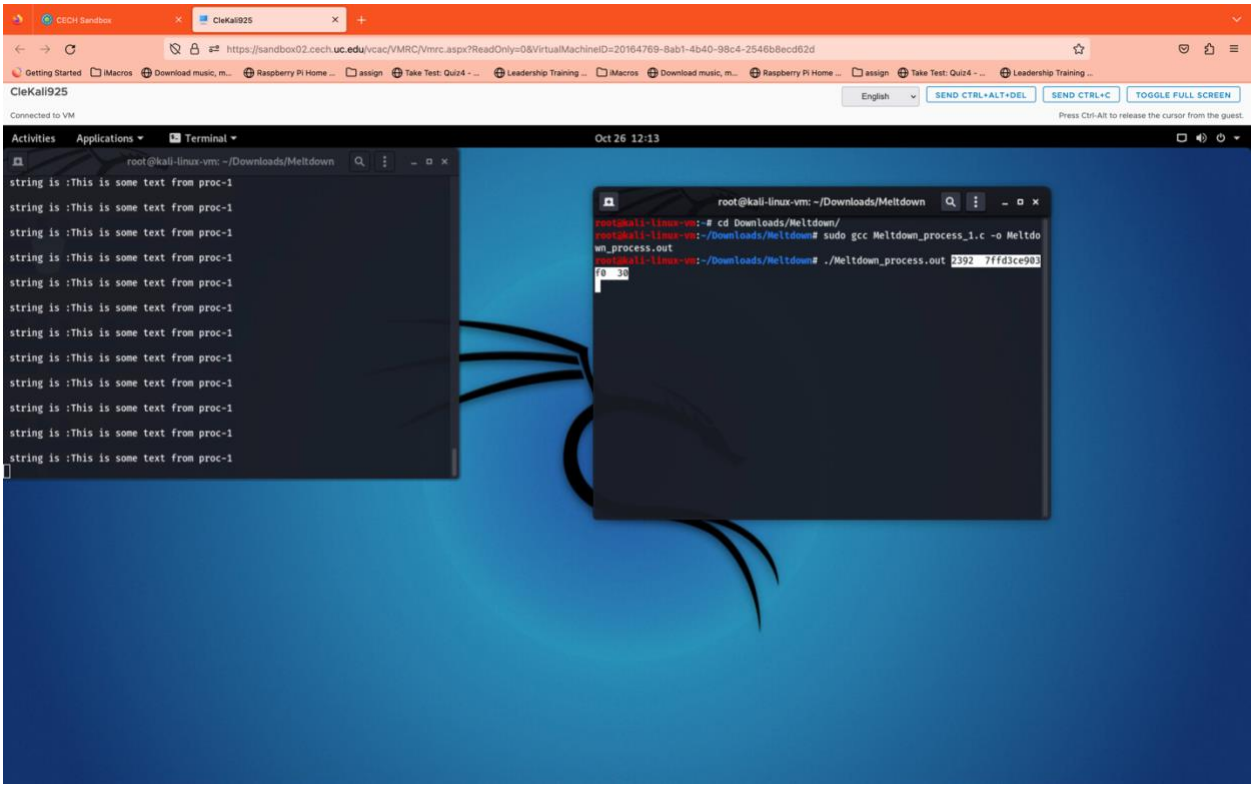
Now on another tab Execute following commands

sudo gcc Meltdown_process_1.c -o Meltdown_process.out

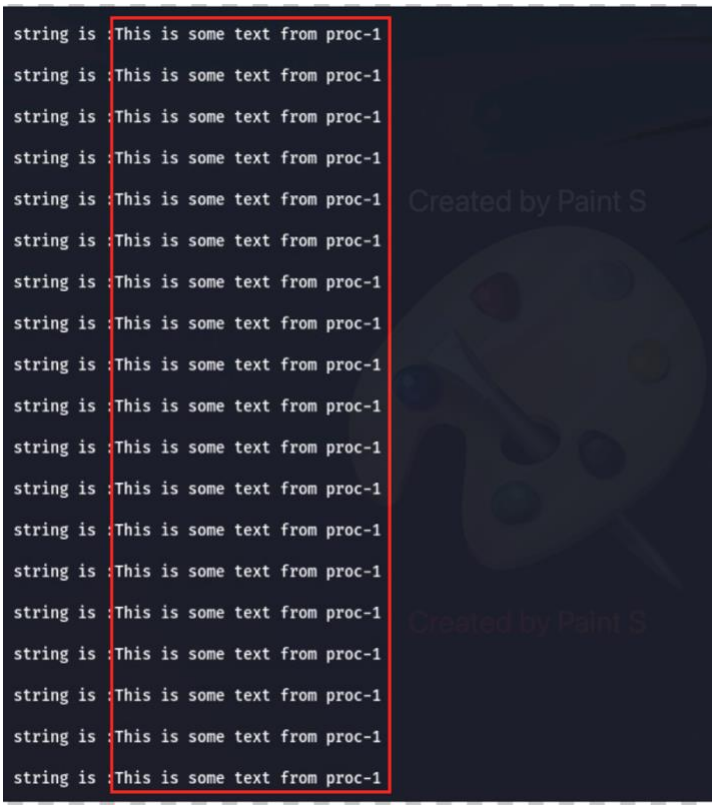


sudo ./Meltdown_process.out <paste the copied string here> **(at this point execute step ./a.out)**

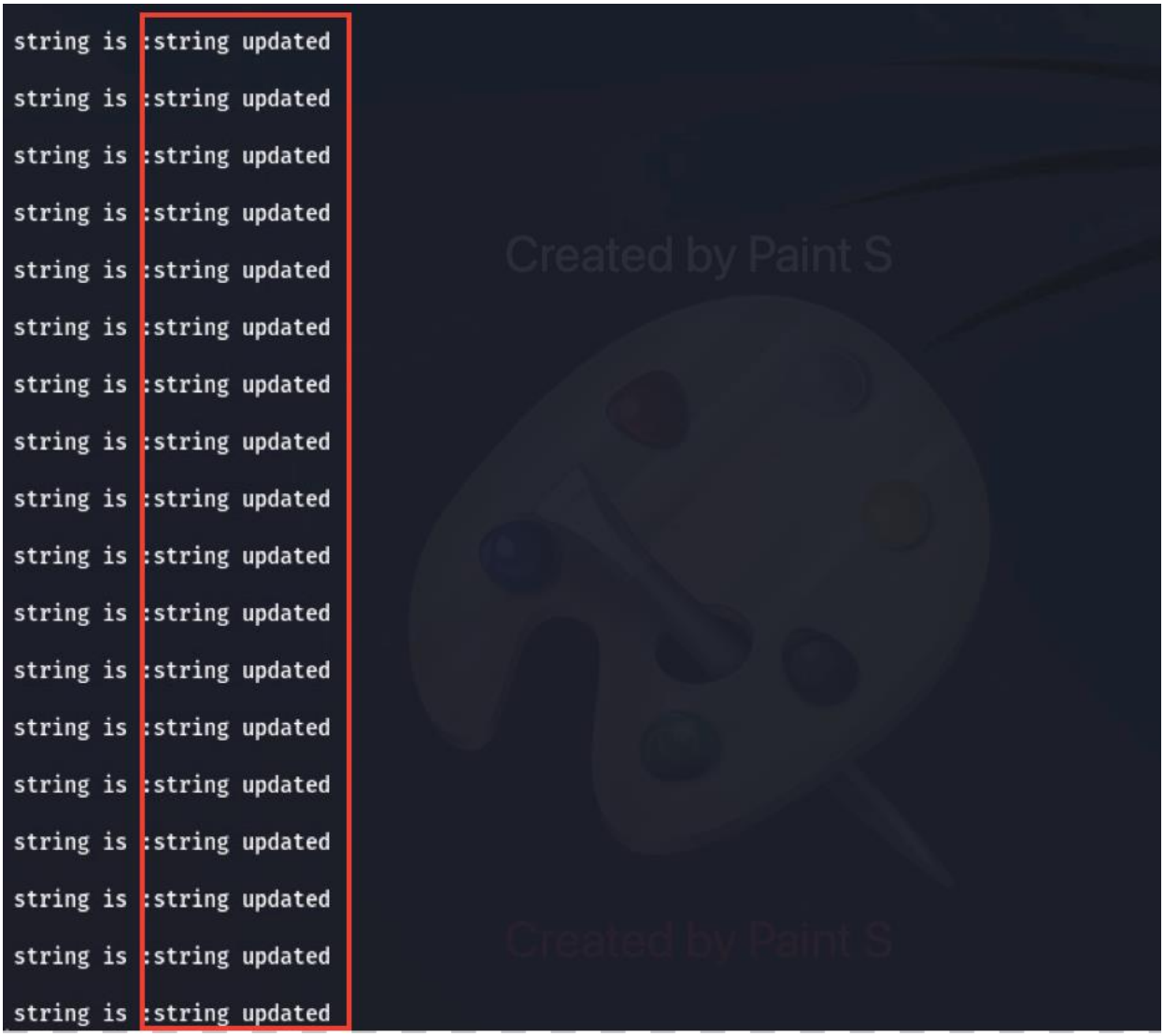




Before running the meltdown process the string getting printed in the output output of proc-1 is as follows:



After running the Meltdown process we can see the string getting printed in the process one(proc-1) is changed.



Question 2: Take a screenshot of the process showing "String is: This is some text from proc-1.

Question 3: Take a screenshot of the process showing "String is: string updated.

Question 4: What is Spectre attack? Please explain.

Answer:

Question 5: What is speculative execution? Please explain.

Answer:

Information about some of the functions we are using in the meltdown process are as follows:

ptrace(PTRACE_ATTACH, pid, 0, 0);

The **ptrace**() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

PTRACE_ATTACH-

Attach to the process specified in *pid*, making it a tracee of the calling process. The tracee is sent a **SIGSTOP**, but will not necessarily have stopped by the completion of this call; use waitpid(2) to wait for the tracee to stop. See the "Attaching and detaching" subsection for additional information. (*addr* and *data* are ignored.)
Permission to perform a **PTRACE_ATTACH** is governed by a ptrace access mode **PTRACE_MODE_ATTACH_REALCREDS** check.

waitpid(pid, NULL, 0):

Suspends the calling process until a child process ends or is stopped. More precisely, waitpid() suspends the calling process until the system gets status information on the child. If the system already has status information on an appropriate child when waitpid() is called, waitpid() returns immediately. waitpid() is also ended if the calling process receives a signal whose action is either to execute a signal handler or to end the process.

pid_t *pid*

Specifies the child processes the caller wants to wait for:

- If *pid* is greater than 0, waitpid() waits for termination of the specific child whose process ID is equal to *pid*.
- If *pid* is equal to zero, waitpid() waits for termination of any child whose process group ID is equal to that of the caller.
- If *pid* is -1, waitpid() waits for any child process to end.
- If *pid* is less than -1, waitpid() waits for the termination of any child whose process group ID is equal to the absolute value of *pid*.

int **status_ptr*

Points to a location where waitpid() can store a status value. This status value is zero if the child process explicitly returns zero status. Otherwise, it is a value that can be analyzed with the status analysis macros described in “Status Analysis Macros”, below.

The *status_ptr* pointer may also be NULL, in which case waitpid() ignores the child's return status.

int *options*

Specifies additional information for waitpid(). The *options* value is constructed from the bitwise inclusive-OR of zero or more of the following flags defined in the sys/wait.h header file:

- WCONTINUED — return the status for any child that was stopped and has been continued.
- WEXITED — wait for the process(es) to exit.
- WNOHANG — return immediately if there are no children to wait for.
- WNOWAIT — keep the process in a waitable state. This doesn't affect the state of the process; the process may be waited for again after this call completion.
- WSTOPPED — wait for and return the process status of any child that has stopped because it received a signal.
- WUNTRACED — report the status of a stopped child process.

pread(fd_proc_mem, buf, len, address)

pread() reads up to *count* bytes from file descriptor *fd* at offset *offset* (from the start of the file) into the buffer starting at *buf*. The file offset is not changed.

pwrite(fd_proc_mem, buf, len, address)

pwrite() writes up to *count* bytes from the buffer starting at *buf* to the file descriptor *fd* at offset *offset*. The file offset is not changed.

External References:

https://spectreattack.com/spectre.pdf
https://meltdownattack.com/meltdown.pdf