

# Network Security - Project 5 - Meltdown Attack

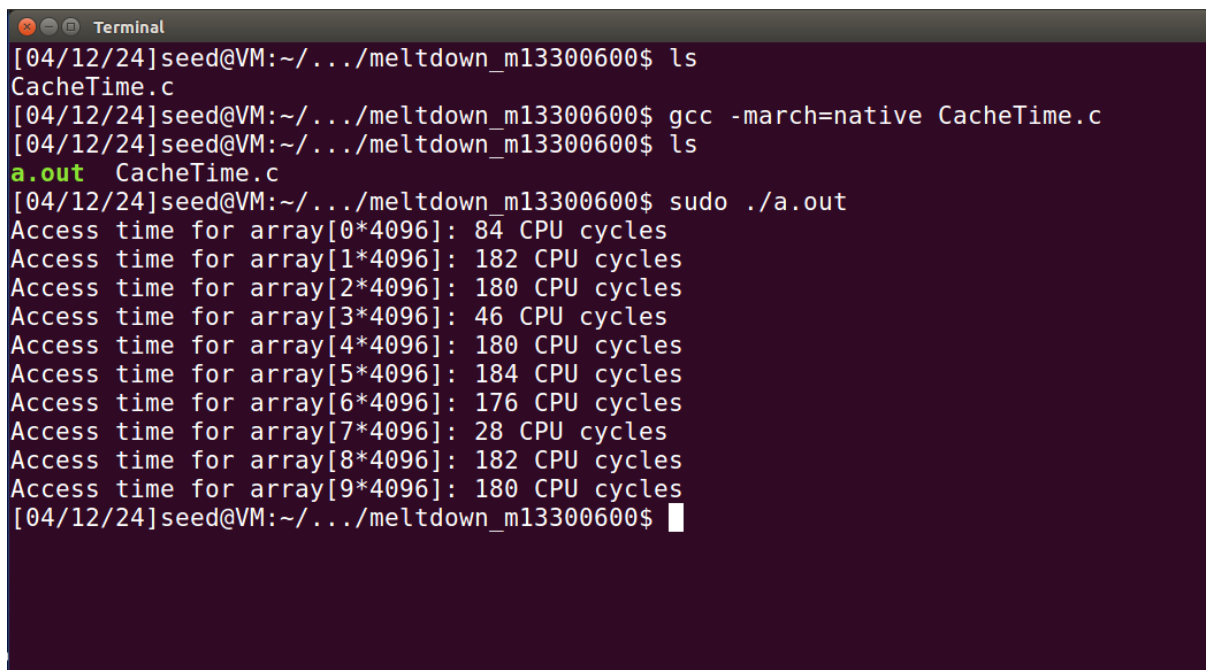
## Task 1: Reading from Cache versus from Memory

Cache memory serves to supply data to high-speed processors at an accelerated rate compared to main memory. Cache memories exhibit considerably faster access times than main memory. To observe this time disparity, we conducted an experiment.

### Compilation

```
gcc -march=native CacheTime.c
ls
sudo ./a.out
```

### Screenshots

A terminal window titled "Terminal" showing the execution of a program. The user is in a directory ~/.../meltdown\_m13300600. They list files, compile CacheTime.c with gcc -march=native, list files again to show a.out, and then run sudo ./a.out. The output shows access times for various array indices in CPU cycles.

```
Terminal
[04/12/24]seed@VM:~/.../meltdown_m13300600$ ls
CacheTime.c
[04/12/24]seed@VM:~/.../meltdown_m13300600$ gcc -march=native CacheTime.c
[04/12/24]seed@VM:~/.../meltdown_m13300600$ ls
a.out  CacheTime.c
[04/12/24]seed@VM:~/.../meltdown_m13300600$ sudo ./a.out
Access time for array[0*4096]: 84 CPU cycles
Access time for array[1*4096]: 182 CPU cycles
Access time for array[2*4096]: 180 CPU cycles
Access time for array[3*4096]: 46 CPU cycles
Access time for array[4*4096]: 180 CPU cycles
Access time for array[5*4096]: 184 CPU cycles
Access time for array[6*4096]: 176 CPU cycles
Access time for array[7*4096]: 28 CPU cycles
Access time for array[8*4096]: 182 CPU cycles
Access time for array[9*4096]: 180 CPU cycles
[04/12/24]seed@VM:~/.../meltdown_m13300600$
```

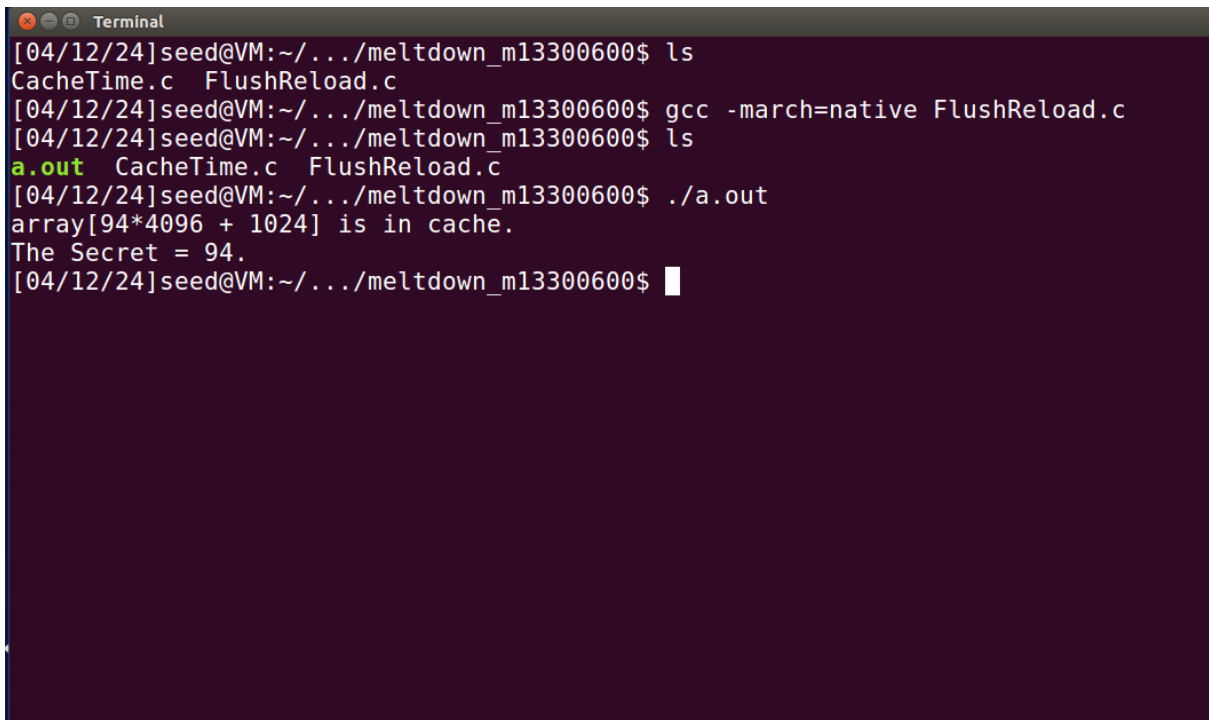
## Task 2: Using Cache as a Side Channel

The objective of this task is to utilize a side channel to extract a secret value employed by the victim function. This technique, known as FLUSH+RELOAD, aims to exploit cache behavior to infer the secret value.

### Compilation

```
gcc -march=native FlushReload.c
ls
sudo ./a.out
```

### Screenshots



```
Terminal
[04/12/24]seed@VM:~/.../meltdown_m13300600$ ls
CacheTime.c  FlushReload.c
[04/12/24]seed@VM:~/.../meltdown_m13300600$ gcc -march=native FlushReload.c
[04/12/24]seed@VM:~/.../meltdown_m13300600$ ls
a.out  CacheTime.c  FlushReload.c
[04/12/24]seed@VM:~/.../meltdown_m13300600$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/12/24]seed@VM:~/.../meltdown_m13300600$
```

## Task 3: Place Secret Data in Kernel Space

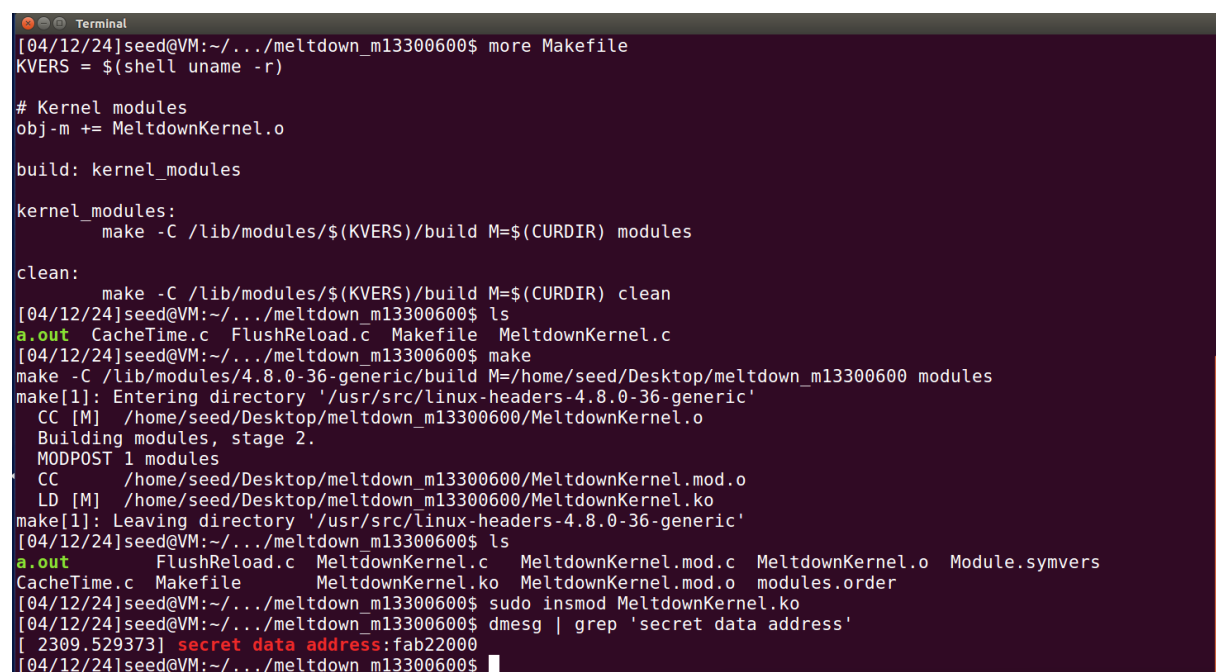
In this task, we take steps to prepare for the Meltdown attack by placing secret data in the kernel space. This allows us to demonstrate how a user-level

program can access and reveal this secret data. To facilitate this demonstration, we employ a kernel module provided in `MeltdownKernel.c`.

## Compilation

```
ls
make
ls
sudo insmod MeltdownKernel.ko
dmesg | grep 'secret data address'
```

## Screenshots



```
Terminal
[04/12/24]seed@VM:~/.../meltdown_m13300600$ more Makefile
KVERS = $(shell uname -r)

# Kernel modules
obj-m += MeltdownKernel.o

build: kernel_modules

kernel_modules:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules

clean:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean
[04/12/24]seed@VM:~/.../meltdown_m13300600$ ls
a.out  CacheTime.c  FlushReload.c  Makefile  MeltdownKernel.c
[04/12/24]seed@VM:~/.../meltdown_m13300600$ make
make -C /lib/modules/4.8.0-36-generic/build M=/home/seed/Desktop/meltdown_m13300600 modules
make[1]: Entering directory '/usr/src/linux-headers-4.8.0-36-generic'
  CC [M] /home/seed/Desktop/meltdown_m13300600/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC /home/seed/Desktop/meltdown_m13300600/MeltdownKernel.mod.o
  LD [M] /home/seed/Desktop/meltdown_m13300600/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.8.0-36-generic'
[04/12/24]seed@VM:~/.../meltdown_m13300600$ ls
a.out  FlushReload.c  MeltdownKernel.c  MeltdownKernel.mod.c  MeltdownKernel.o  Module.symvers
CacheTime.c  Makefile  MeltdownKernel.ko  MeltdownKernel.mod.o  modules.order
[04/12/24]seed@VM:~/.../meltdown_m13300600$ sudo insmod MeltdownKernel.ko
[04/12/24]seed@VM:~/.../meltdown_m13300600$ dmesg | grep 'secret data address'
[ 2309.529373] secret data address:fab22000
[04/12/24]seed@VM:~/.../meltdown_m13300600$
```

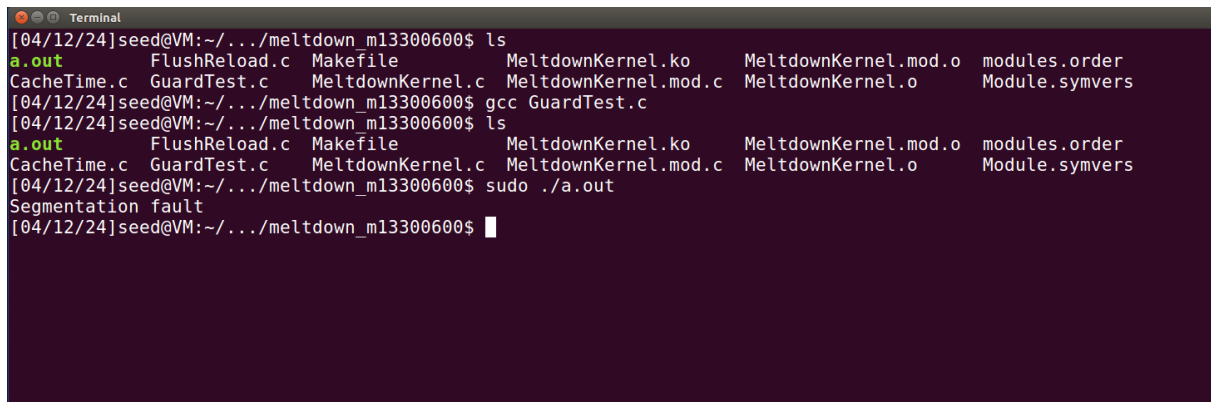
## Task 4: Access Kernel Memory from User Space

In this task, we aim to access kernel memory from user space, leveraging the address of the secret data obtained in the previous step. We will conduct an experiment to ascertain whether we can successfully retrieve the secret data directly from its kernel-space address.

## Compilation

```
gcc GuardTest.c
ls
sudo ./a.out
```

## Screenshots



```
Terminal
[04/12/24]seed@VM:~/.../meltdown_m13300600$ ls
a.out      FlushReload.c  Makefile      MeltdownKernel.ko  MeltdownKernel.mod.o  modules.order
CacheTime.c  GuardTest.c    MeltdownKernel.c  MeltdownKernel.mod.c  MeltdownKernel.o      Module.symvers
[04/12/24]seed@VM:~/.../meltdown_m13300600$ gcc GuardTest.c
[04/12/24]seed@VM:~/.../meltdown_m13300600$ ls
a.out      FlushReload.c  Makefile      MeltdownKernel.ko  MeltdownKernel.mod.o  modules.order
CacheTime.c  GuardTest.c    MeltdownKernel.c  MeltdownKernel.mod.c  MeltdownKernel.o      Module.symvers
[04/12/24]seed@VM:~/.../meltdown_m13300600$ sudo ./a.out
Segmentation fault
[04/12/24]seed@VM:~/.../meltdown_m13300600$
```

## Task 5: Handle Error/Exceptions in C

In Task 4, we observed that attempting to access kernel memory from user space can lead to a program crash. However, in the context of the Meltdown attack, it's essential to handle such errors gracefully rather than letting the program terminate abruptly. One approach to handle errors in C is by defining custom signal handlers to capture exceptions raised by catastrophic events.

## Compilation

```
gcc ExceptionHandling.c
sudo ./a.out
```

## Screenshots

```
Terminal
[04/12/24]seed@VM:~/.../meltdown_m13300600$ ls
a.out      FlushReload.c  MeltdownKernel.c  MeltdownKernel.mod.o  Module.symvers
CacheTime.c  GuardTest.c  MeltdownKernel.ko  MeltdownKernel.o
ExceptionHandling.c  Makefile  MeltdownKernel.mod.c  modules.order
[04/12/24]seed@VM:~/.../meltdown_m13300600$ gcc ExceptionHandling.c
[04/12/24]seed@VM:~/.../meltdown_m13300600$ sudo ./a.out
Memory access violation!
Program continues to execute.
[04/12/24]seed@VM:~/.../meltdown_m13300600$
```

## Task 6: Out-of-Order Execution by CPU

The concept of out-of-order execution plays a crucial role in modern CPU architectures, enabling enhanced performance by allowing the processor to execute instructions beyond the strict sequential order specified by the program. This task delves into the intricacies of out-of-order execution and its implications on program behavior, particularly in scenarios involving memory access violations.

### Compilation

```
gcc -march=native MeltdownExperiment.c
sudo ./a.out
```

### Screenshots

```
Terminal
[04/12/24]seed@VM:~/.../meltdown_m13300600$ gcc -march=native MeltdownExperiment.c
[04/12/24]seed@VM:~/.../meltdown_m13300600$ ls
a.out      FlushReload.c  MeltdownExperiment.c  MeltdownKernel.mod.c  modules.order
CacheTime.c  GuardTest.c  MeltdownKernel.c  MeltdownKernel.mod.o  Module.symvers
ExceptionHandling.c  Makefile  MeltdownKernel.ko  MeltdownKernel.o
[04/12/24]seed@VM:~/.../meltdown_m13300600$ sudo ./a.out
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[04/12/24]seed@VM:~/.../meltdown_m13300600$
```

## Task 7.1: The Basic Meltdown Attack - A Naive Approach

In this task, we aim to exploit the race condition created by out-of-order execution to steal a secret from the kernel memory. Building upon the FLUSH+RELOAD technique introduced earlier, we modify our approach to target specific kernel data rather than arbitrary array elements.

## Compilation

```
gcc -march=native MeltdownExperiment.c  
sudo ./a.out
```

## Observations:

- The attempt to access kernel memory at the specified address led to a memory access violation, as expected.
- The signal handler successfully caught the SIGSEGV signal raised by the memory access violation.
- Despite the unsuccessful attempt to steal the secret from the kernel memory using the FLUSH+RELOAD technique, the program continued execution without crashing.

## Analysis:

- The FLUSH+RELOAD technique was ineffective in stealing the secret from the kernel memory due to the protection mechanisms in place preventing unauthorized memory access.
- The memory access violation demonstrates the robust security measures implemented to safeguard kernel memory from unauthorized access by user-space programs.
- Despite the failure of the attack, the program's ability to handle the memory access violation gracefully highlights the importance of error handling mechanisms in ensuring program stability and reliability.

## Screenshots

```
Terminal
[04/12/24]seed@VM:~/.../meltdown_m13300600$ gcc -march=native MeltdownExperiment.c
[04/12/24]seed@VM:~/.../meltdown_m13300600$ sudo ./a.out
Memory access violation!
[04/12/24]seed@VM:~/.../meltdown_m13300600$
```

## Task 7.3: The Basic Meltdown Attack - Using Assembly Code to Trigger Meltdown

To improve the success rate of the Meltdown attack, we introduce additional lines of assembly instructions before accessing the kernel memory. These instructions are designed to occupy the CPU's execution units with useless computations, increasing the likelihood of successfully stealing the secret from the kernel memory.

Despite varying the loop count in the assembly instructions within the `meltdown_asm()` function, the success of the Meltdown attack remained consistent across different loop counts. Regardless of whether the loop count was set to 20, 200, 400, or even 2,000,000, the attack consistently managed to steal the secret from the kernel memory.

### Observations:

- The Meltdown attack was successful in all scenarios, indicating that the additional lines of assembly instructions effectively exploited the CPU's speculative execution to leak the secret from the kernel memory.
- Varying the loop count within the assembly instructions did not significantly impact the attack's success rate. Despite changing the duration of useless computations, the attack remained successful in all cases.
- This consistency suggests that the effectiveness of the Meltdown attack may be less influenced by the duration of useless computations and more by the ability to exploit speculative execution to access privileged memory.

### Compilation

```
gcc -march=native MeltdownExperiment.c
```

## Screenshots

```
[04/12/24]seed@VM:~/.../meltdown_m13300600$ gcc -march=native MeltdownExperiment.c  
[04/12/24]seed@VM:~/.../meltdown_m13300600$ sudo ./a.out
```