Mission 2: Shhh... It's a Secret (in the Cache)

Aakash Tarang

June 25, 2025

1 Introduction

This will be my solution report for the second week assignment of the learner's space Hardware Security course.

The following describe my implementation details

2 Creating a shared memory address

I achieved this using the same way as the video tutorial discussed

```
const int SIZE = 4096;//size of shared memory segment
const char *name = "OS";//name of shared memory segment
int fd;//file descriptor id, similar to an open file
char *shared_mem;
fd = shm_open (name, O_CREAT|O_RDWR, 0666) ;
ftruncate(fd, SIZE);
shared_mem = (char*)mmap (0, SIZE, PROT_READ|PROT_WRITE,
    MAP_SHARED, fd, 0);
```

3 Cache-Based Covert Channel Communication

The following functions implement a covert communication channel by exploiting CPU cache timing side effects. The sender modulates cache states to transmit data, while the receiver infers bits by measuring memory access latency.

3.1 Function `send_bit(char* addr, int bit)`

This function transmits a single bit by manipulating the cache state of a shared memory location (`addr`).

- **If** `bit == 1`**:**
  - Calls `maccess(addr)` to bring the memory line into the cache.
  - The receiver observes a **fast access** (cache hit), inferring a logical '1'.
- **If** `bit == 0`**:**

  - Skips `maccess(addr)`, leaving the cache line evicted.
  - The receiver observes a **slow access** (cache miss), inferring a logical '0'.

- `flush(addr)` is done when the bit is zero, so that the latency measurement by the receiver program indicates longer time interval. The receiver program also contains a flush which prepares the chache line for the next bit.
- `usleep(BIT_TIME_US)` enforces synchronization between sender and receiver by introducing a fixed delay.

3.2 Function `send_byte(char* addr, char byte)`

This function transmits a full byte by iteratively calling `send_bit()` for each bit in `byte`.

- Iterates over each of the 8 bits (LSB to MSB) in `byte`.
- Extracts the current bit using `(byte >> i) & 1`.
- Calls `send_bit(addr, bit)` to transmit the bit via cache state modulation.

3.3 Receiver Operation

A receiver reconstructs the transmitted data by:

- Monitoring `addr` and measuring access latency.
- Interpreting **fast accesses** (cached) as '1' and **slow accesses** (uncached) as '0'.
- Reassembling received bits into bytes.

3.4 Security Implications

This method is a classic example of a **timing-based side-channel attack**, with applications in:

- Exfiltrating data across privilege boundaries (e.g., Spectre attacks).
- Covert inter-process communication (e.g., malware communication).

Mitigations include cache partitioning, constant-time programming, and aggressive cache flushing.

4 Synchronization Pattern

A sync pattern is descibed at the begining of the sender.c file

```
#define SYNC_PATTERN 0b10101010
```

This sync pattern is transmitted first as, it's used as a synchronization signal so the receiver knows when the actual message starts. Without this, the receiver might misinterpret noise or timing mismatches.

5 Transmit the Message

```
for (int i = 0; i < msg_size; i++) {
    send_byte(&shared_mem[OFFSET], msg[i]);
}
```

Here, the sender loops through each character in the message stored in msg[]. Each character is a byte, and it gets sent using the same send_byte() function. That function breaks the byte into 8 bits and sends each bit using Flush+Reload.

So if your message was "Hi", this loop would send the ASCII value of 'H', then 'i', each broken into bits. and we finally end the message with the following flag.

6 Reciever Code

Bit Detection using Flush+Reload

The receiver detects transmitted bits by leveraging cache access timing differences via the `Flush+Reload` side-channel. The core of this detection logic is implemented in the `detect_bit` and `receive_byte` functions:

```
int detect_bit(char *addr) {
    usleep(BIT_TIME_US / 2);
    unsigned long long start = rdtsc();
    maccess(addr);
    unsigned long long end = rdtsc();
    flush(addr); // flush for next bit
    return (end - start) < THRESHOLD ? 1 : 0;
}
```

Explanation:

- `usleep(BIT_TIME_US/2)`: Waits for a fixed bit interval to align with sender's transmission.
- `rdtsc()` before and after `maccess(addr)` measures access latency.
- If the memory is in cache (access is fast), it indicates the sender accessed it — representing a bit `1`.
- If not in cache (access is slow), it indicates the sender did not access it — representing a bit `0`.
- `flush(addr)`: Evicts the memory address from all cache levels for the next bit.

The `receive_byte` function assembles 8 such bits into a byte:

```
char receive_byte(char *addr) {
    char byte = 0;
    for (int i = 0; i < 8; i++) {
        int bit = detect_bit(addr);
        byte |= (bit << i);
```

```
    }
    return byte;
}
```

It sequentially calls `detect_bit()` for each bit of the byte and shifts the result into position. This forms a complete ASCII character, enabling reconstruction of the original message sent by the sender.