Solution to Exercise #4

|  |  |
| --- | --- |
| Authors | |
| **Name** | **Matriculation Number** |
| Beshoy Saad | 2572741 |

# Secure Over the Air Update

1. The server from which the file is downloaded can be compromised (i.e. an attacker can gain access to the server and change the file contents). In this case, even if a secure connection over HTTPS is used, the file is not guaranteed to be authentic, as the secure connection only protects the file from being tampered with while in transit from server to client.
2. The update process works as follows: initially, flash memory is divided into a number of segments called partitions. The first partition is called “factory” and it contains the original firmware written to the device via e.g. USB. Then come a number of OTA partitions (OTA0, OTA1… etc.) that contain images received via the OTA upgrade mechanism. To give an example, let’s consider an ESP32 that has 3 partitions: factory, OTA0, and OTA1. When the chip boots for the first time, the image residing in “factory” is running. Then this running code can start the OTA process: it first finds a free flash partition (let’s say it is OTA0), then starts downloading the new image from the specified server piece by piece and writing each piece to the chosen flash partition OTA0. Once the whole new image is in the chosen partition, it is checked for validity and if the check passes, OTA0 is marked as the new boot partition. The next time the chip is reset, the bootloader will try to boot from OTA0 first. One drawback for this method might be the needed flash storage requirement (at least 2x the size of the firmware).
3. See attached code. **Note: when building the project please link against the stable release (v3.2.2) of the esp-idf library not the master branch.**
4. At least two things could go wrong with this method:
   1. An attacker might switch the legitimate firmware with a malicious one once the signature verification is done but before the second download begins, resulting in the bootloader overwriting the chip’s firmware with attacker code.
   2. The update process could be interrupted for any reason (power loss for example), leaving the chip in an unusable state, as the old firmware was overwritten and the new firmware wasn’t completely downloaded from the server or written to flash. Having a fallback mechanism is crucial in OTA upgrade systems.

# No Buffer Overflow, We Checked It!

1. Speculative execution is a feature built into CPUs to increase performance. This feature enables the CPU to fetch and execute instructions whose results might or might not end up written back to memory and used in the rest of the program. This can happen in branches (if-then-else for example), where the CPU doesn’t wait for the result of the conditional statement to be calculated, but instead starts fetching and executing one of the branches (selected according to some algorithm). If the conditional statement ends up leading to the branch just executed, then this technique has just saved us some CPU cycles. If not, then the results of the speculative execution are discarded, and the other branch is fetched and executed instead. This works thanks to pipelining, which is the ability of a CPU to execute multiple independent instructions at the same time; for example fetching from memory and decoding an instruction. So in order not to leave the pipeline empty, the CPU simultaneously calculates the conditional statement and speculatively fetches and executes one of the branches in the hope it’s the correct one.

Caching is the process of keeping a copy of data or instructions accessed recently by the CPU in a memory structure that is faster and has much shorter access time than RAM, called cache. Modern CPUs usually have a 3-level cache structure, each level larger but slower than the last (going from L1 to L3, which is nearest to farthest from the CPU). When a CPU tries to fetch a memory location, it first looks it up in cache. If it’s not present, then it’s fetched from RAM and stored in cache, and if the cache is full then some other data is evicted (least recently accessed, oldest… etc.) The reason caching is used in many modern architectures is to increase performance, as there’s a bottle neck at the CPU-memory interface due to difference in speeds.

1. Spectre is a micro-architectural attack that abuses speculative execution to read from memory locations in the user space by utilizing the fact that data that was loaded into cache as a result of speculative execution will remain in cache after the processor recovers to the correct branch and thus access time to this data will be less than un-cached data.

Meltdown, on the other hand, abuses out-of-order execution to leak kernel memory to user space via a covert channel that consists of allocated memory pages. The attack utilizes exceptions raised when trying to access kernel memory from user space.

The code snippet presented is a variant of Spectre, where the attacker supplies a value for readFrom that is out of range of the array string (larger than 420 in this case). If stringLength is uncached while readFrom is cached, the processor will begin executing the code inside the conditional block while stringLength is being fetched from RAM. Now the first statement in the conditional block unsigned char c = string[readFrom]; stores data that is out-of-bounds of the array string into the variable c. This is user-space data that shouldn’t be visible outside the victim process, and up to this point it is still safe, since the CPU is going to discard those changes once the value of stringLength is retrieved from RAM. The next line assigns one of two values to the variable indexUncachedData: 0 if the least significant bit of c is zero, or 900 if the least significant bit of c is 1. The next line accesses the array uncachedData at index indexUncachedData, which has the effect of loading the memory location pointed to by uncachedData[indexUncachedData] into cache. This data remains in cache even after the CPU recovers from the speculative execution miss, which constitutes a side-channel that depends on measured access time to detect which data is cached and which is not. Since the rest of the array uncachedData is not in cache, an attacker can – after the conditional block terminates – try to access uncachedData[0] and uncachedData[900] and measure the access time for both. If the access time for uncachedData[0] indicates it’s in cache, then the least significant bit of the variable c must have been 0, which means that the least significant bit of the memory location string[readFrom] must be 0 as well. On the other hand, if the access time indicates that uncachedData[900] is cached, then the least significant bit of string[readFrom] must be 1. Thus, by executing this code once, an attacker is able to leak 1 bit of user-space memory from the victim process.

It is important to note that uncachedData must be initially in RAM and not in cache in order for the attack to work, because the attack depends on measuring access time to know which index of the array was accessed and thus placed in cache. If the array was already in cache when the attack begins, both indices 0 and 900 would have short access time and the attacker wouldn’t be able to tell which index was accessed.

# Dolly - The Unclonable Sheep

1. According to Tehranipoor et al. (2015)[[1]](#footnote-1), DRAM can be used as a PUF since it exhibits power-up randomness that is a result of manufacturing imperfections, similar to SRAM. The explanation for the randomness lies in the fact that when the DRAM is powered up, the individual capacitors in each memory cell are neither charged nor discharged. This leaves them at , which is the bias voltage of the memory cell’s transistor. Manufacturing imperfections will, however, make the capacitor tend towards either or , which results in a memory value reading of 1 or 0, respectively. These startup values can thus be used as a fingerprint for device identification, since DRAM is used in constructing main RAM modules of most modern computers.
2. 1. We developed a sketch to allocate 256 bytes on the stack and print the allocated (but uninitialized) data to the serial port. As expected, seemingly random data was in the buffer, but running the sketch multiple times, it becomes clear that the data is not much different between runs. The sketch was run on the Arduino 10 times, generating 10 different buffers. In order to determine a fingerprint, we needed a way to measure how close a particular buffer is to the “average” buffer; i.e. we needed a buffer that didn’t have too many ones or zeroes compared to any buffer you’d get by running the sketch on our Arduino. We tackled this problem by taking each buffer and calculating the sum of all of its bytes, and then we calculated an average sum by adding the individual sums and dividing by 10. We then determined which buffer of the 10 to use as a fingerprint by calculating the difference between each buffer’s sum and the average sum. In our case, buffer #9 had the least difference, so we decided to use it as a fingerprint.
   2. See attached code. Board Serial Number: 85431303636351F02151.

1. Tehranipoor, F., Karimian, N., Xiao, K., & Chandy, J., "DRAM based intrinsic physical unclonable functions for system level security", In Proceedings of the 25th edition on Great Lakes Symposium on VLSI, (pp. 15-20). ACM, 2015 [↑](#footnote-ref-1)