JavaScript Sidechannel Cache Attack

Arturo Perez
Professor Zhang
CS 4285
15 December 2018

I. Introduction

Side-channel analysis over time has proven to be a reliable cryptanalytic technique which can be used to extrapolate private information hidden inside a host and its processes. It does this by analyzing physical signals that a device emits (i.e. heat. power, etc.) as a byproduct of the device performing secure computations.

For my project I will use a JavaScript cache attack. After a victim opens the malicious webpage, the website will execute a JavaScript based cache attack, which lets the attacker track accesses to the victim's last-level cache over time (L3 Cache in Intel Processors). Since this single cache is shared by all CPU cores in specifically Intel Processors this access information can provide the attacker knowledge about the victim and the system under attack.

II. Implementation

I began by creating a "fake" music recommendation web application where I can lure a user to open it. This simple web page was implemented using HTML/CSS and fully interacts with the Spotify API to actually generate song recommendations to the user. In this way, a user can be tricked to thinking they are browsing a safe website.

The meat and potatoes of my attack lies in index.html under the function initSideChannel(). After my demo yesterday, I went back and rewrote most of my attack as I realized that I need to allocate large arrays to interact with cache memory. We now make use of the Prime and Probe technique outlined in *The Spy in the Sandbox* paper. However, I must also point out that my implementation is a vast oversimplification of the whole algorithm described in the paper. Therefore my attack hardly infers anything secretive about the user, but we will see that I do in fact manage to flush out the L3 cache as measured by access latency time.

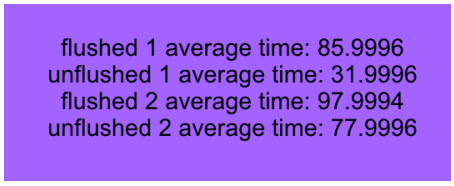
We begin by initializing two huge arrays of approximately 8MB such that this will flush the L3 cache of the victim's computer. I then declare a variable x that I will repeatedly flush from cache and measure the access latency time to access that variable. If the access latency time is low, then the variable is still in cache. If the access latency time is high, then we know that the variable is not in cache anymore and is being accessed from somewhere else in memory (most likely RAM).

III. Results and Conclusion

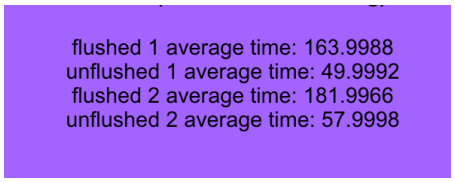
When someone goes to visit my website, they will notice that there is a slight loading time before they are able to type an artist into the query. This loading time is the JavaScript attack taking its course, constantly flushing the cache and measuring the access time for the variable X.

The averages of the access times are printed to the page after execution of the JavaScript. Interestingly enough, one can see that there is indeed a vast difference in access time between the flushed vs. unflushed variable. Here are some examples from when I ran it on my computer:

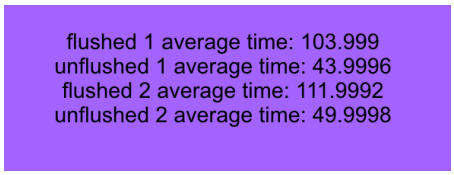
Example A:



Example B:



Example C:



As is shown, access latency times are vastly different depending on whether the variable is in cache or somewhere else in memory. This confirms that I am properly flushing the cache, which is the first step in being able to infer information from the victim.

I have implemented the basics of Prime and Probing, yet I am still quite far from being able to actually steal information from a victim. *The Spy in the Sandbox* outlines further techniques to identify "interesting" cache regions as well as function calls which may trigger interesting behavior but it honestly would take someone more experienced than I to fully implement these features. Regardless, I enjoyed my time on this project.