Detecting Page Swap from JavaScript

Nicholas Starke | https://twitter.com/nstarke | https://github.com/nstarke

I saw a blog post come across twitter over the weekend (http://blog.skylined.nl/20161118001.html) in which the author mentions the following as an aside:

> Side note: did you know you could make a pretty good estimate of the amount of RAM installed on a system by timing large memory allocations? Simply have some Javascript allocated several megabytes of RAM over and over and when it suddenly slows down, that means the OS is swapping memory to disk. You can then see how much you have allocated, make some assumptions about how much RAM was used by other applications and the OS and calculate the amount of RAM installed pretty accurately in my experience…

So I decide to test it out. I was testing on a pair of MacBook Pros, both with 16GB of RAM. The result was positive; I was able to detect via timing analysis when memory reached ‘swap time’ — when memory was paged out to swap on disk. My proof of concept, included later in this post, was tested on Chrome, Firefox, Brave, and Safari. Chrome seems to work flawlessly, while Firefox and Safari work well with a few false positives.

There were a couple of obstacles to overcome on OSX. First of all, I couldn’t simply write a constant value to memory; OSX has the concept of compressed memory. When a constant value is written in continuous memory, the operating system compresses that data. So the first obstacle was writing random data to memory. Seems simple enough, I can use Math.random() to generate somewhat (not cryptographically strong) random values that will suffice for my purposes.

Since I am allocating data at the megabyte level, looping over the entire memory space and calling Math.random() at each iteration takes a long time. To keep the feedback cycle as tight as possible, I iterated on the length of the allocated memory space divided by eight, and then generated a random value for each iteration. That random value was then used with various arithmetic operations against the iteration index in order to produce different values for the other eight values between iterations. This makes it difficult for the operating system to compress, since the data is seemingly random for each eight byte iteration-block.

At this point we have an in memory data structure that is sufficiently random and all we need to do is keep creating additional copies of this data structure until we detect a major timing difference in how long it takes to allocate the memory. Different browsers seem to allocate memory differently; for instance, Chrome seems to allocate memory as soon as the ArrayBuffer is created. Firefox on the other hand doesn’t always allocate memory when the ArrayBuffer is created; it appears to allocate memory when values are written to the ArrayBuffer.

When measuring the timing, I kept Activity Monitor running so I could watch the RAM allocations. There was consistently a 5–6x slowdown in allocation time around the period of time when memory pressure went up and the Swap size started increasing. Extensive testing showed that six times slower seemed to be the magic number.

During testing, I noticed very few false positives. False positives occurred when the allocation time increased over six times the average allocation time before memory was exhausted. Conversely, before I had the magic

number six worked out, I noticed a few false negatives, where the condition which detected the swap failed to execute and the JavaScript kept allocating memory until the operating system prompted me to destroy the process because memory and disk swap were both exhausted. Once I had calibrated the magic number six, I saw very few false positives and no false negatives.

Using the optimizations above, I was able to reach swap time, detect swap time, and restart the test process (without reloading the HTML document) in under 2 minutes on both machines I tested.

So in the end, what is this technique useful for? What value does detecting the amount of free RAM have for an attacker? It could be used to monitor when a computer is in use, as when the free RAM amount changes frequently it is safe to assume programs are being loaded, used, and unloaded from memory. Conversely, when the RAM amount doesn't change by much over a long period of time, it might indicate that the computer is not in use. And without further adieu, here is the proof of concept code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Tetris Memory Allocation</title>
  </head>
  <body>
    <script>
      var startTime = window.performance.now();
      var master = [];
      var counter = 0;
      var interval = 16;
      var bufferLength = 1024 * 1024 * interval;
      var total = 0;
      function test(){
        var timer = setInterval(function(){
          var start = window.performance.now();
          var buffer = new ArrayBuffer(bufferLength);
          var view = new Int32Array(buffer);
          var amount = Math.floor(view.length / 8);
          for (var i = 1; i < amount; i++){
             var index = i * 8;
             view[index] = Math.floor(Math.random() * bufferLength);
             view[index - 1] = view[index] + i;
             view[index - 2] = view[index] - i;
             view[index - 3] = view[index] * i;
             view[index - 4] = view[index] << i;
             view[index - 5] = view[index] >> i;
             view[index - 6] = view[index] | i;
             view[index - 7] = view[index] & i;
          }
          var end = window.performance.now();
          var timeLength = (end - start);
          master.push(view);
          counter++;
          total += timeLength;
          var average = total / counter;
          if (timeLength > average * 6) {
```

```
            console.log('Paged to disk, took ' + ((window.performance.now()
- startTime) / 1000) + ' seconds with ' + (counter * interval) + ' mb
allocated - ' + interval + 'MB interval');
            clearInterval(timer);
            delete master;
            master = [];
            total = 0;
            counter = 0;
            startTime = window.performance.now();
            // restart
            setTimeout(test, 150);
          }
        }, 150);
      }
      // kick off testing
      test();
    </script>
  </body>
</html>
```