Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2020,7(10):74-79

Research Article ISSN: 2394-658X

Debugging Memory Leaks in Enterprise Web Applications: A Case Study of Optimum Processes

Tanmaya Gaur

Email id - tanmay.gaur@gmail.com

ABSTRACT

Memory Leaks are a notorious problem known for impacting application servers running complex backend applications. Memory leaks are alsorampantwith web applications, especially complex and interactive single-page applications (SPAs) that rely on bulkyframeworks like AngularJS. These Web Application Memory leaks degrade the performance, usability, and reliability of web applications, and can even cause them to crash or freeze. With longer duration or faster leak rate, the memory build-upcan impact the browser running these web applications, an event which can expandthe impact to other web applications running on the browser.

While the paper will provide references to some popular tooling available to developers, The intent of this paper is not to deep dive into the tools for debugging. This paper will instead focus on the best process to apply in the debugging lifecycle. There are many resources on the internet about toolsthat help debug memory utilization on a browser. Thereis however minimal documentationon the ways to approach a web application memory leak in a structured algorithmic fashion. The process and approach were questions we were forced to research for when solutioning a complex memory leak with a telco's abnormally largeCRM application.

This paper describes the process and best practices that were successfullyutilized foranalysis of the memory usage patterns, identify root cause of memory leaks, and implement effective solutions to improve memory efficiency and user experience. The techniques and learnings discussed can be applied to most enterprise or non-enterprise web application.

Keywords: Memory Leak, Web Development, Operations, Debugging, CRM

INTRODUCTION

For Telco's, a CRM and CSM applications enables telecom customer service representatives to support sales and to perform various service tasks related to account and device management such as payment processing, order fulfilment and troubleshooting.

In our sampled scenario, this CRM was a web application built using Angular JS, a popular JavaScript framework that provides a declarative and modular way of creating dynamic and responsive web applications. The CRM was used by agents to assist calling customers one after the other during their workday and hence had session timings often exceeding multiple hours. The application was extensively used during this time. The application users were constantly running into a situation where the application sessions crashed with an out of memory error message (Chrome's famous Aw, Snap!) while they were still working with a user. Similar issues were experienced on other web browsers running the app. They had to relaunch the application post the crash. This was significantly impactful as their application state was lost during the crash as well. E.g. if they were in the middle of a sale, all the information on the user and hard and soft goods added to cart was lost during the crash. This also resulted in bad experience for the calling end customers of the telco.

The fact that this application was developed Single Page App style had a significant role in such situation. AngularJS simplifies the development of Single Page App style applications by providing features such as data binding, directives, services, filters, routing, and dependency injection. However, Single Page Appimplementations like Angular if used for applications with a long and active user sessioncan become an issue if coupled with even a minor memory leak. The minor leakcan slowly build up over timeand lead to a degraded experience. A degradation is a situation where we start exhausting available resources.

So, what is a memory Leak? A memory leak occurs when memory that is no longer needed by the application is not released back to the system. Once this unreleased memory build-up starts exhausting available memory, itresults in poor application performance. Once all available memory is exhausted, application eventually crash or freeze. While not our focus, we will dive a little more detail about the role CPU, I/O, Physical and Virtual memory must play in the subsequent section.

Memory leaks are particularly problematic for complex enterprise web applications with the following characteristics:

[1]. Applications that are large and complex, consisting of hundreds of components, views, and services that interact with each other and with external APIs.

[2]. Applications that are long-lived, meaning that users typically keep them open for extended periods of time and perform multiple transactions and navigations within the same session.

[3]. Applications that are used in diverse and constrained environments, such as Citrix servers with shared resources, WebView on mobile devicesetc., where memory and CPU limitations apply have more pronounced and noticeable impacts.

[4]. Applications that have high quality and performance expectations, as they are critical for the business operations and customer satisfaction of the enterprise.

Now that we have the context setup, let’s talk about how the rest of the paper is structured.

[1]. First part of the paper will dive deeper into memory leaks and the application behaviors associated with memory issues. We will also discuss some other browser resource constrictions which show similar symptoms and may often be consumed with memory leaks.

[2]. Next section listspopular tools and supplementary resources which can be used to measure and monitor memory usage for UI applications.

[3]. Subsequent section will present a triage process, this isthesteps and algorithm that we refined to identify and resolve our memory leak problem. This approach involves several steps which are to be repeated to get to a solution.

[4]. The subsequent section summarizes the lessons learned from this experience.

BROWSERS AND MEMORY LEAKS

As defined earlier, a memory leak is a situation where a web application (or script) fails to release memory it is no longer using. This eventually leads to the device or application running out of available memory required for it to function which causes experience degradation.

Memory thresholds can be at the device, browser or tab level depending on the device(see Figure 1). An example would be iOS WebView which apply severe limitations on memory available to the tab. There is additional complexity with clients like Citrix VMs which may be sharing underlying memory across multiple user sessions and thusavailable memory at a given time is constantly changing.

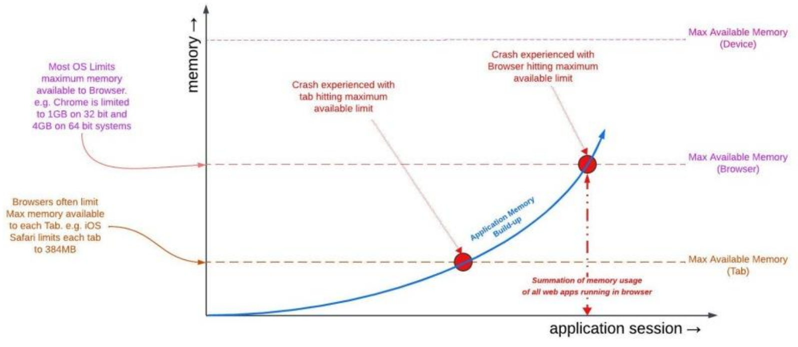

memory →
Max Available Memory (Device)
Most OS Limits maximum memory available to Browser. e.g. Chrome is limited to 1GB on 32 bit and 4GB on 64 bit systems
Crash experienced with tab hitting maximum available limit
Crash experienced with Browser hitting maximum available limit
Max Available Memory (Browser)
Browsers often limit Max memory available to each Tab. e.g. iOS Safari limits each tab to 384MB
Application Memory Build-up
Max Available Memory (Tab)
Summation of memory usage of all web apps running in browser
application session →

Figure 1: Memory build-up (x) due to Leak against session duration (y)

Another aspect determining total available memory is Virtual Memory. Virtual memory is a memory management technique that uses hardware and software on a device to compensate for physical memory shortages in a computer's operating system. The technique moves data from RAM to disk storage which increases the capacity of the main memory in a process called “swapping”.While virtual memory is a function provided by the operating system, browsers often utilize this capability in different ways. Hence the total available memory is at times determinant on the combination of operating system and browser being used. If your memory issue only replicates on specific browser, this may be a useful insight to remember.

Memory leaks mostly happen due to poorly written Java-script code or add-on extensions running on the browser. There are some other issues that can cause memory leak like experience degradation but have unrelated causality.

[1]. Operations with large disk I/O or heavy CPU usage can cause resource exhaustion resulting in latency.

[2]. There may be instances where large number of parallel running apps or an unoptimized web app with high memory requirements can lead to memory being exhausted without a leak.

TOOLING ESSENTIAL TO DEBUG A MEMORY LEAK

Most web browsers now-days come built in with some great tooling to understand and triage an application's memory footprint. Chrome dev-Tools are a favourite among web developers for this purpose. The available capabilities range from being able to view basic CPU and memory statistics for each Tab to be able to visualize memory consumption in real-time with a performance profiler. Once you have a handle on the flow you intend to focus on, techniques like heap snapshots allow you to better understand allocations and even look for detached DOM Nodes, a common scenario which results in unreleased memory.
A detached DOM is a scenario where an element is no longer attached to the Document Object Model (DOM) tree but is still referenced by some JavaScript running on the page. The browser is not able to collect the detached element because of the reference.
A powerful method which currently has limited browser support is the window.performance.memory property which returns the estimated memory being used by the page. We have had significant success in logging this property from page java-script at run-time alongside web user activity to determine what actions negatively impact the memory profile (memory allocated but not released when action is complete). Do remember that this protocol is not widely supported across browsers right now.
Other browsers like Firefox also provide similar tooling with the different memory views. It gets tricky with iOS app web-views where the only available options require you to run apps in debug mode alongside xCode for troubleshooting.
There is also theWindows Performance Recorder toolprovided by windows operating system which allowed us to run detailed analysis at the operating system memory profiles. The advantage of using this tool is that it remains completely non-intrusive to the web application and the browser.

RECOMMENDED TRIAGE PROCESS

This section will go over a recommended process to go about debugging a web application memory leak (refer figure 2). While not set in stone, the goal of this process is to organize and structure your debugging method and to be able to use data driven techniques to get to a result.

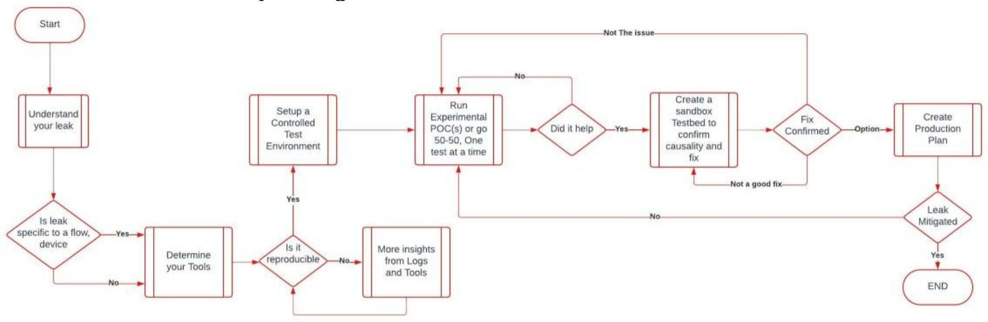

Start
Understand your leak
Is leak specific to a flow, device
Yes
No
Determine your Tools
Is it reproducible
Yes
No
More insights from Logs and Tools
Setup a Controlled Test Environment
Run Experimental POC(s) or go 50-50, One test at a time
No
Did it help
Yes
Create a sandbox Testbed to confirm causality and fix
Not a good fix
Fix Confirmed
Not The issue
Option
Create Production Plan
No
Leak Mitigated
Yes
END

Figure 2: Triage process for debugging, isolating, and remediating a memory leak.

A. Understand Your Leak

Is the leak with all browsers or specific ones? Determining if the leak is ties to specific operating system, browser or user flow will significantly alter your process and choice of tools.
A browser app can appear to slow down, crash or freeze for multiple reasons. It is advisable to determine your issue category as it alters your debugging process. Some examples

[1]. Device running out of physical Memory. E.g. in one scenario, an incorrectly configured background device scan was using significant memory leading to the device crash. The agents actively using the Web applications experienced the impacts as their browser freezing and giving an out of memory error.

[2]. Browser out of memory because multiple applications running across various tabs consume all available memory.

[3]. Sometimes bad code can result in CPU spikes where all the CPU available to your process is exhausted resulting in the tab/browser freezing. At times, it becomes difficult to distinguish this from a memory issue. These however can almost always be tied to trigger of a specific culprit code or step and are not caused over time.

B. Reproduce the Steps

While ideal, this is not always easily achievable. Users of your application complain about application slowness, freeze, or crash which points towards a memory leak, but it often becomes difficult to reproduce. This is a resultant of multiple factors like unavailability of production like data, lack of production behavior or the leak being related to a specific edge case scenario. It is often vital to look at the users who are facing the issue and looking for patterns in their activity, their roles and application usage etc. Any system logs or actual conversations if possible are helpful in looking for these patterns.

C. **Capture Systematic Insights**

Certain Browsershave started supportingthe performance.memory property which can be observed and logged. This property can allow you to log the browser's memory footprint at specific time stamps or post a relevant activity block which can then be observed to identify how certain user actions or session timing impact the leak. This could lead to patterns and insights towards the reproduction steps.

There are certain browser extensions which allow you to implement similar capabilities of observing user actions and browser health. It is however not always possible to deploy a browser extension depending on the nature of your website and userbase.

D. **Understand your non-intrusive Tooling**

Often the tools used for debugging a memory leak end up negatively impacting the application behavior and memory patterns. These tools can often time create new and unrelated challenges which end up wasting focus of the team debugging the original leak. In our debugging, we found that chrome's memory profiler tools often negatively impacted the size and causes of our memory leak.

E. **Controlled Test Lab Setup**

Complex applications often have a lot of things going on. Depending on the user credentials, roles and flow, the applications may behave different. To accurately measure the problem and impact of possible solutions being tested, it is invaluable to create a control test environment. This refers to creating a platform that replicates real-world conditions for deploying and testing solutions while limiting variability from external or internal factors. In our case, this meant.

[1]. Creating a synthetic script which executed a pre-configured sequence of application browsing steps using the same test data.

[2]. Locking down the environment from any interferences like code drops external to work being done on memory leak triage.

[3]. Creating virtualization by creating stubbed API responses based on the target's performance, data and dynamic behavior expected of the API. This eliminates need for a live backend while also liberating the debugging process from test data and other backend dependencies.

We first ran our synthetic scripts on our application against the stubbed backend and then re-ran the same after each experimental POC (item vii) was individually dropped into the environment. We monitored the impacts or changes to the memory profile of the application before and after each drop to determine success.

F. **Isolate your leak, Go 50-50**

It is advisable to isolate the memory leak as much as possible. E.g. If you are experiencing an issue with a commerce checkout flow, is the issue only prevalent when the user is utilizing the upper funnel (product selection all the way till cart) or lower funnel (checkout up to order confirm). While this sounds simplistic, this is significant help as it reduces the number of variable elements you must try and debug and optimize.

In our scenarios, we achieved this by dropping aspects of functionality iteratively and verifying if the leak still exists. If the leak goes away at a particular iteration, the removed functionality and associated code is related to the issue, if not, we repeat.

G. **Lot of Experimental POC(s)**

It is useful to ideate on areas you believe may be contributing towards the memory leak. Once there is a list of areas to focus, came up with a list of options to prove or disprove the memory leak as originating from that specific area of the application. These were deployed and tested as proof of concept (POC) in our controlled environment.

We ended up conducting 40 POC(s) which ranged from small like completely dropping reference to third party tagging (e.g. google analytics for behavior monitoring) to as large as updating our angular version across our application stack. For each fix, we tabulated the data (Refer figure 3)to create a data driven matrix to help us decide if we want to move forward with the fix or not.

H. **The 3-snapshot Technique**

A google recommended approach when using the profiler, it utilizes the chrome built in Heap profiler's Summary view. It requires the users to go to Profiles and take a heap snapshot before an intensive part of your code kicks in and then one more after this code has been allowed to run for a while. Repeat. You can then compare the snapshots using the Summary (and other) views to confirm what allocations have been made along with their impact on memory. This technique proved useful. It is important to point out that unlike the runtime memory profiler, the heap snapshot was a less intrusive technique and did not impact or change the memory characteristics of our leak.

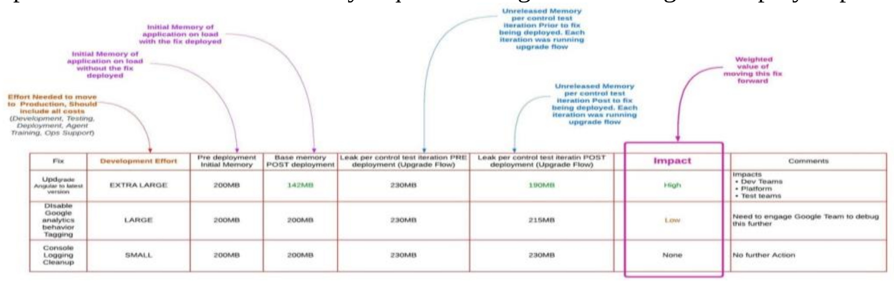
I. **Confirm your Leak in an isolated test-bed**

Once you have a good idea of the area which is an issue, try reproducing the leak by replicating only that code and function outside your application. In our scenario, we essentially write a hello world angular app with the culprit pattern to verify if the leak recreates without the other modules of our application.

J. **Data driven decision-making**

Once you have results, deciding if these fixes are worth pursuing further. What determines the value of a fix can be based on multiple factors depending on the application, user-base and impacts. In our case, this was determined on a combination of the how the fix impacts the memory leak (Refer figure 3) alongside the actual effort required to implement in production.

e.g. Although upgrading Angular provided significant relief, the effort required to implement it in production made it equivalent in value to a tagging update that removed unnecessary tags. The tagging update provided minimal relief but only required configuration changes to deploy to production.


Effort Needed to move to Production, Should include all costs (Development, Testing, Deployment, Agent Training, Ops Support)
Initial Memory of application on load without the fix deployed
Initial Memory of application on load with the fix deployed
Unreleased Memory per control test iteration Prior to fix being deployed. Each iteration was running upgrade flow
Unreleased Memory per control test iteration Post to fix being deployed. Each iteration was running upgrade flow
Weighted value of moving this fix forward

Fix	Development Effort	Pre deployment Initial Memory	Base memory POST deployment	Leak per control test iteration PRE deployment (Upgrade Flow)	Leak per control test iteratin POST deployment (Upgrade Flow)	Impact	Comments
Updgrade Angular to latest version	EXTRA LARGE	200MB	142MB	230MB	190MB	High	Impacts • Dev Teams • Platform • Test teams
Disable Google analytics behavior Tagging	LARGE	200MB	200MB	230MB	215MB	Low	Need to engage Google Team to debug this further
Console Logging Cleanup	SMALL	200MB	200MB	230MB	230MB	None	No further Action

Figure 3: Tabulate your results against criticaldecision factors

CONCLUSION

Creating a controlled test bed and conducting controlled tests, one at a time works wonders. However, speed during this process of elimination is crucial.

In our use-case, we conducted 40+ POCs during a month and a half, often rewriting significant parts of our application. The 3-snapshot technique is an amazing tool at helping confirm findings and weeding out false positives. Out of the 40 POCs we tested, 15 POCs displayed significant improvements and wereoperationalized and moved to production. This remediated the situation successfully (Refer figure 4)

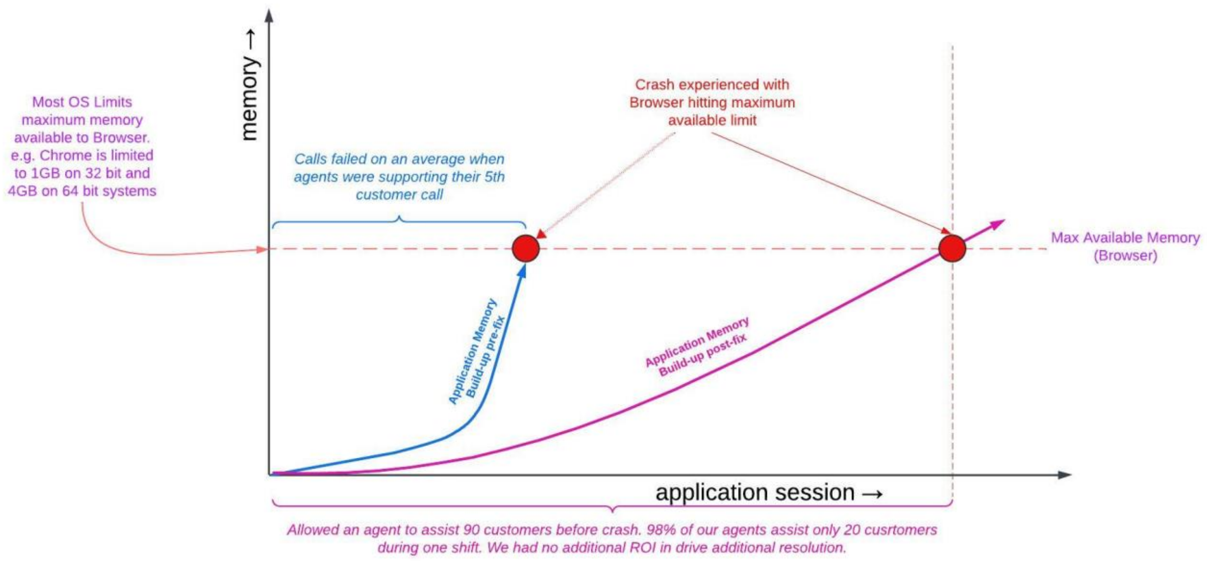


Figure 4: Fix impact plotted against Original Leak

The last but equally important aspect I want to leave you with is knowing when to Stop!
In many cases, minor leaks exist with underlying frameworks and third-party libraries. These result in minimal build-up over time which is immaterial in the bigger scheme of things. As an example, once we had resolved our bigger memory issues, we were left with an angular elements memory leak that resulted in a 5MB build-up every angular element load/unload. While seeming problematic, for a customer service agent who launched 6-8 elements every customer service call and took 40 calls over a period of 8-hour shift, this resulted in a build-up of 1600MB over that period. Still significantly smaller than the total available 4GB of memory on their desktop. The out of box memory leak with the element would have been costly to debug and resolve with no business benefits as we no longer had an application crash issue across the customer service agent shift. Even though the leak was not fully resolved, thiswas a good place to stop.

Conflicts of interest
The author declares that there is no conflict of interest regarding the publication of this paper.

REFERENCES

[1]. The Medium Website article by Joao Santos, 2019. [Online]. Available: https://medium.com/swlh/effective-javascript-debugging-memory-leaks-75059b2436f6
[2]. The Mozilla Website. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Performance/memory
[3]. The Wikipedia website. [Online]. Available: https://en.wikipedia.org/wiki/Memory_leak
[4]. The addyosmaniwebsite. [Online]. Available: https://addyosmani.com/blog/taming-the-unicorn-easing-javascript-memory-profiling-in-devtools/