PRODPERFECT

How ProdPerfect Avoids Impacting Web Performance

testdouble

The following is an analysis of the ProdPerfect secure Javascript snippet by TestDouble.

The core of ProdPerfect's system is proprietary machine learning technology that takes live user data and turns it into actionable code and scripts. To do this, though, ProdPerfect needs a regular stream of user interaction from your site. This is handled through the ProdPerfect tracking script, embedded into your site's HTML through a small snippet.

But a script injected into your site has the potential to create a classic observer problem. If the user's actions are being observed, is the observation itself causing the user to behave differently? There could be many possible reasons for the observation to cause a change in behavior. These potential issues include UI interruptions or interactions and performance issues with the application, both of which can be accidental or intentional side effects of any script inserted into your application.

ProdPerfect works tirelessly to ensure that our script will not impact your user's experience of your application. By making use of industry leading techniques and continuously working to further minimize and eliminate even the smallest performance issues, ProdPerfect has created a script that can observe your users' actions without interruption or other impact.

In this effort, ProdPerfect has identified and adhered to three principles, creating a foundation of high performance even on low-powered devices.

1. Never delay your application's script load or execution
2. Never delay or interrupt the user while monitoring their actions
3. Never delay your application's network requests

Each of these principles encompass a multitude of optimizations within the ProdPerfect script. And all of them are designed with the goal of providing the highest level of service with the least impact to the end user, possible.

Never Delay Your Application's Script Load or Execution

The first opportunity a script has to interrupt a user's experience of a page, is when the script loads, is parsed and executed.

When a browser sees a <script> tag in your application's HTML, it will stop almost all of the other work that it is currently doing and evaluate that script. If there's a "src" to set a URL for the script, it has to be downloaded before it can be parsed. But the script must be parsed for the browser to continue with most of its remaining work. Failing to do that can result in unpredictable results in how the script interacts with the HTML on the page, and other scripts that may need to be downloaded.

The most common method of working around this problem when loading third party scripts, such as ProdPerfect, is to put the <script> tag at the end of the HTML content. This allows the browser to do as much work as possible before loading and executing the script, resulting in an overall improvement in the user experience. The page appears to load faster, and the user is less likely to abandon the application or site entirely.

ProdPerfect recommends you take advantage of this when placing the code snippet for loading the ProdPerfect script. But having the script tag at the end of your HTML is only one part of the optimization process. The second part of the optimization is to embed as small a script as possible directly in the HTML, and not use a "src" url to load the initial script.

By inserting a script, shown in Code Listing 1, that is approximately 1KB in size, when minimized, your application will be able to kickstart the process of loading ProdPerfect's full script in as little time as possible.

Code Listing 1: Example ProdPerfect JavaScript Snippet

```
!function(e,r,n){var t,o=!!window.Keen&&window.Keen;n[e]=n[e]||{ready:function(r){var a,d=document.
getElementsByTagName("head")[0],i=document.createElement("script"),c=window;i.onload=i.
onreadystatechange=function(){if(!(i.readyState&&!/^c|loade/.test(i.readyState)||a)){if(i.onload=i.
onreadystatechange=null,a=1,t=c.Keen,o)c.Keen=o;else try{delete c.Keen}catch(e){c.Keen=void 0}
n[e]=t,n[e].ready(r)}},i.async=1,i.src="keen-tracking.min.js",setTimeout(function(){d.parentNode.
insertBefore(i,d)})}}}("ProdPerfectKeen",0,this),ProdPerfectKeen.ready(function(){var e=new
ProdPerfectKeen({projectId:"(your project id)",writeKey:"(your write key)",requestType:
"beacon",host:"your_company.datapipe.prodperfect.com/v1"});e.extendEvents({visitor:{user_
id:null}});e initAutoTracking({ignoreDisabledFormFields:!1,ignoreFormFieldTypes:
["password","email"],recordClicks:!0,recordFormSubmits:!0,recordInputChanges:!0, recordPageViews:!
0,recordPageUnloads:!0,recordScrollState:!0})}); initAutoTracking({ignoreDisabledFormFields:!1,
ignoreFormFieldTypes: ["password","email"],recordClicks:!0,recordFormSubmits:!
0,recordInputChanges:!0,recordPageViews:!0,recordPageUnloads:!0,recordScrollState:!0})});
```

There will be no network traffic for this script. The time to parse and execute is minimized. And this small script allows the full ProdPerfect script to be downloaded, parsed and executed in the background by taking advantage of the third step in optimizing the load and parse steps.

Embedded into your HTML, this script will do two things. First, it creates a new <script> tag and sets the "src" attribute to the full ProdPerfect script location. Then, after the page load event has fired, the embedded script uses a setTimeout call to insert the new script tag into the document.

As shown in Figure 1, Having the setTimeout in place allows the browser to deprioritize loading ProdPerfect's full script. Any page load scripts that need to run within your full application can take over immediately and the browser will schedule the insertion of the ProdPerfect script tag for a later time. From the blue rectangle on the right hand side of figure one, Figure 2 then zooms in to show the delayed script call to insert the new <script> tag into the Document.

Figure 1. Timing for Script Download and Parsing

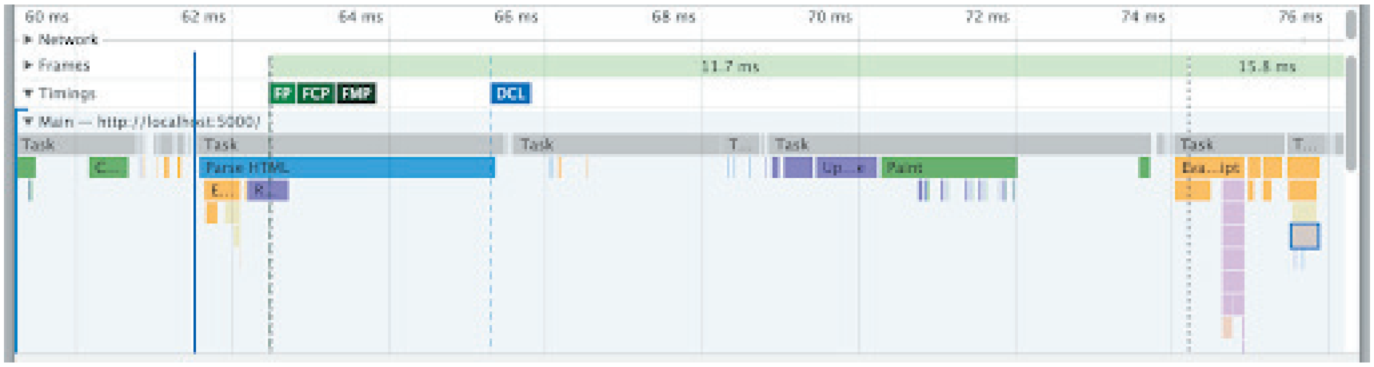
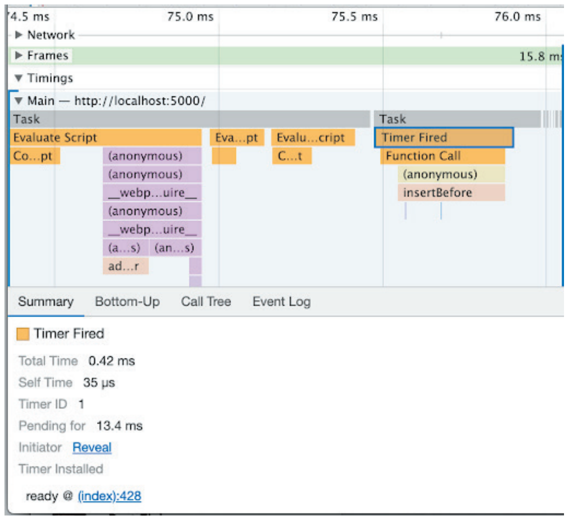


Figure 2. Delayed Script Insertion



Timer Fired

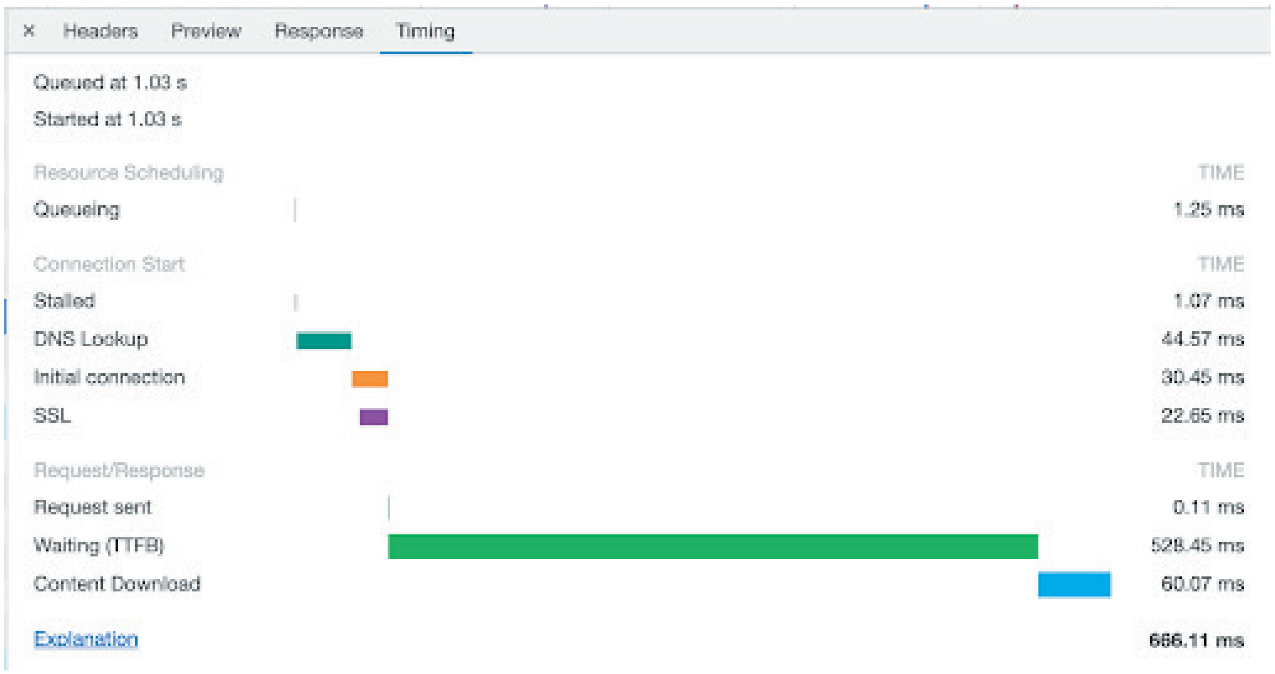
Total Time 0.42 ms
Self Time 35 µs
Timer ID 1
Pending for 13.4 ms
Initiator Reveal
Timer Installed

ready @ (index):428

The final piece of the optimization puzzle, for loading and parsing the ProdPerfect script is to use an industry leading content delivery network ("CDN") to ensure the script is delivered as quickly as possible, to the browser, no matter where they are in the world.

Figure 3 shows an initial download of the full ProdPerfect script under less than ideal network conditions. The time it takes to receive the file is mostly comprised of network lag. Figure 4, then, shows reloading the same page and allowing the browser's cache to do what it was built to do.

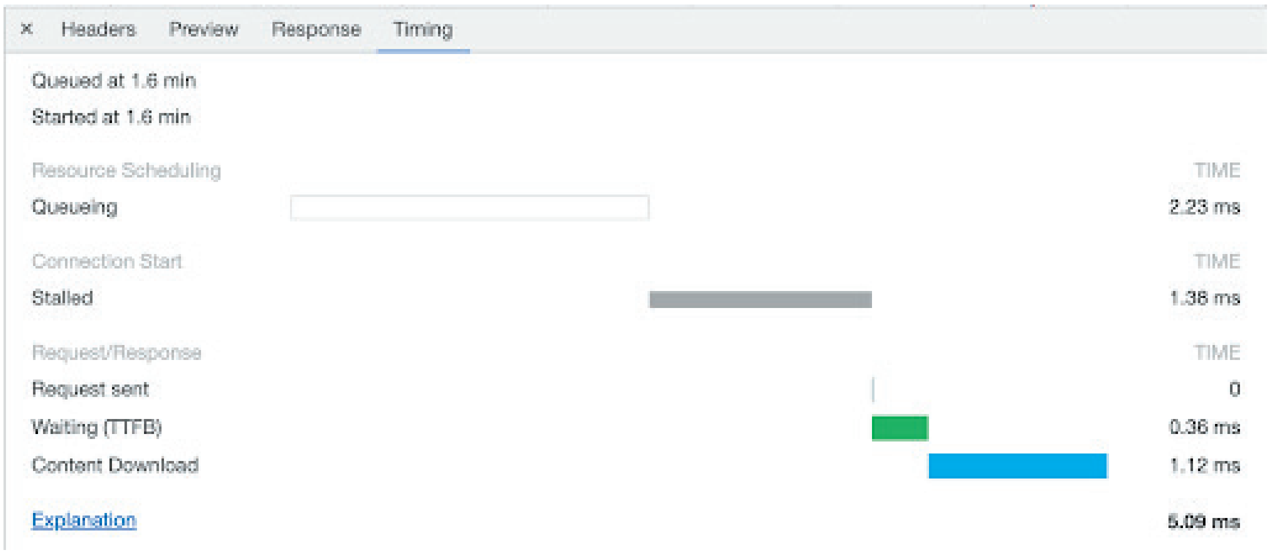
Figure 3. Initial ProdPerfect Script Download


× Headers Preview Response Timing

Queued at 1.03 s
Started at 1.03 s

Resource Scheduling	TIME
Queueing	1.25 ms
Connection Start	**TIME**
Stalled	1.07 ms
DNS Lookup	44.57 ms
Initial connection	30.45 ms
SSL	22.65 ms
Request/Response	**TIME**
Request sent	0.11 ms
Waiting (TTFB)	528.45 ms
Content Download	60.07 ms
Explanation	666.11 ms

Figure 4. Cached ProdPerfect Script Loading


× Headers Preview Response Timing

Queued at 1.6 min
Started at 1.6 min

Resource Scheduling	TIME
Queueing	2.23 ms
Connection Start	**TIME**
Stalled	1.38 ms
Request/Response	**TIME**
Request sent	0
Waiting (TTFB)	0.36 ms
Content Download	1.12 ms
Explanation	5.09 ms

The time savings in this example is well over half a second - easily enough to ensure any page beyond the user's landing page will begin executing ProdPerfect's script as quickly as possible while still taking advantage of the delayed <script> tag insertion as discussed above.

Optimization of downloading, parsing and executing a script is only the first step to ensuring the user's experience of your application is not impacted. Once the ProdPerfect script executes, it must also stay out of the user's way while they work with your application.

Never Delay or Interrupt a User's Actions

Most JavaScript applications use event handlers such as "click" events or "change" events in form elements by centering the code around individual DOM elements. A <button> that has a click event handler, for example, could set up a simple listener as shown in Code Listing 2

Code Listing 2. Simple Button Click Event Handler

```
myButton.addEventListener('click', function(e) {
  // ...
});
```

When dealing with a small number of events and code that has specific knowledge of what actions each event should cause, this works out fine. However, dealing with a large number of events - especially events that share the same event handler - can quickly turn into a performance problem. Too many event handlers will destroy performance on your page, due to the high cost of DOM interactions and the DOM event system in JavaScript.

Performance problems begin with even a moderate number of DOM elements and event handlers. They becomes unbearable to the user when every DOM element in the app must have a click handler attached to it, slowing pages down to a crawl.

This creates a significant challenge. Your page may have five or ten thousand DOM elements, and ProdPerfect must handle a "click" event on every single element. If ProdPerfect had to look at all DOM elements in your page and attach an event handler to each - such the code shown inCode Listing 3 - the performance of the page would crawl to a halt and users would abandon your site entirely.

Code Listing 3. Unsafe Code to add Event Handlers on All DOM Elements

```
document.querySelectorAll('*').forEach(function(element) {

  element.addEventListener('click', function(e) {
    // ...
  });

});
```

How does ProdPerfect handle the need to track all "click" events, and other DOM element events without impacting the user experience, then?

By taking advantage of the DOM's event bubbling, ProdPerfect is able to create a single event handler for any given event type, such as "click". Once a user clicks on an element within the DOM, the event will bubble its way up to the root document where the event handler exists. From there, the JavaScript event handler code can determine which DOM element was clicked and correctly track that information.

The same is true for other events as well, such as the "change" event on form elements. Each type of event has a single event handler, meaning there will be a finite and deterministic number of event handlers every time ProdPerfect is brought into a page. Code Listing 4 shows a simplified example of both a single "click" and single "change" event handler that would receive events from any DOM element and form element, respectively.

Code Listing 4. A Single Click Handler for all DOM Elements

```
document.addEventListener('click', function(e){

  // get the specific element that was clicked
  clickedElement = e.target;

  // … work with the element here
});
```

Optimizing the event handlers for a large DOM element count is only one step toward the efficiency that ProdPerfect's tracking script requires, however. The data collected must be sent to the ProdPerfect servers, and this introduces a new set of challenges.

Never Delay your Application's Network Requests

When JavaScript applications first became common in web development, communication with a server was limited to only XMLHttpRequests. While this added an incredible amount of capability to web pages, it also created some new and difficult situations to accommodate when working in more advanced applications.

If an application needs to send data to a server before unloading itself, there have been relatively few choices in the past. You could send an XMLHttpRequest and hope that the server received it before the page unloaded and cancelled the request. Or you could use other hacks, such as inserting an 'img' tag into the DOM and set the source url to a server endpoint where you can pass request parameters, as shown in Code Listing 5.

Code Listing 5. Using an IMG Tag For Posting Data

```
// create an image tag
const img = document.createElement("img");

// set the src to include data the server wants
img.src = "/path/to/server/endpoint?data=1234&foo=7890&bar=czasdfwqer";


// insert the img tag, forcing the DOM to wait for the network call to complete
document.getElementsByTagName('body').appendChild(img);
```

In the case of XMLHttpRequests, you have no guarantee that the server communication will complete before the page unloads. Additionally, your request is going to be queued alongside every other request from the application. If you're using a script that should provide background services only, such as the ProdPerfect tracking script, this can introduce unwanted side effects in the user experience. Too many requests queued up will cause delays in network round trips and possibly make the UI of the application appear to be laggy or unresponsive.

With the image hack, the problem of unloading a page before the request completes is solved. However, the same network disadvantages as an XMLHttpRequest will be present. This trick will also add the disadvantage of forcing the DOM to load yet another element which has to be accounted for in CSS and other scripts, to ensure it is not visible and not causing any errors in the JavaScript console. Furthermore, this technique will directly harm the user experience by causing a delay between the current page and the page to which the user is trying to navigate, due to the nature of DOM elements loading content.

Neither of these "solutions" – plain XMLHttpRequests, nor image elements – is acceptable. There are other workarounds and fallbacks, still, but many of these can create additional problems for both the users and the developers of the application.

What is the answer, then? A script such as ProdPerfect needs to run in the background, track user actions, and send information back to a server. It requires a less intrusive method of server communication. One that meets these expectations:

- Uses the lowest priority possible, allowing other requests to execute first
- Queues up in the background and guarantees execution – even when the current page has completely unloaded
- Executes only when the browser has sufficient resources, so as not to interrupt the user's experience of the application

Enter, the Beacon API – a native JavaScript API set that handles network communication in a manner that satisfies all of the needs of a background script, as listed above.

The ProdPerfect script takes full advantage of the BeaconAPI to ensure network calls to the ProdPerfect servers are not disrupting the user's experience in your application. And in the off-chance that a user is on an older browser that does not support the BeaconAPI, ProdPerfect's tracking script will fall back to the other methods and workarounds, as available. This ensures support for the widest range of browsers, possible.

Your Application Will Not Suffer with ProdPerfect

Through these combined principles and the myriad of code level optimizations that have been put in place, ProdPerfect's script does everything possible to stay out of the way. Your application will experience little to no impact in performance. This ensures users will remain happy as they continue to work with your application and platform.

It's already a win-win situation. And with ProdPerfect's continuous advancements and optimizations for performance, it will only get better from here.