JavaScript Global Namespace Pollution

JavaScript Global Namespace Pollution - pdp, gnucitizen.org.

- Published: date not stated
- Original: <https://www.gnucitizen.org/blog/javascript-global-namespace-pollution/>
- Preserved from: https://www.gnucitizen.org/blog/javascript-global-namespace-pollution/ (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

JavaScript Global Namespace Pollution

Thu, 07 Feb 2008 10:11:56 GMT

by pdp

If you are reading this you are probably thinking what does this post has to do with security. Well, let me explain. One of the ways to detect JavaScript malware is to check for namespace pollutions symptoms. Simply put, if the JavaScript execution container contains more objects then the expected, something wrong is going on. This post will briefly walk through some ideas currently circulating in my head.

Namespace pollution checks are very trivial to perform. The check should be performed from a safer location such as outside of the execution sandbox or somewhere on the top before and after the user input is taken into consideration. The check is very simple really. All that needs to be done is to compare the list of registered objects with the expected list of objects. If they defer, the namespace has been polluted by something. The check can be performed by a function similar to the one discussed by the Atom database over here:

```
function walkJSON(j, c) {
   for (var i in j) {
      c(i, j[i]);

      if (j[i] instanceof Array || typeof(j[i]) == 'object') {
         arguments.callee(j[i], c);
      }
   }
}
```

The function is very simple as you can see, though you have to be careful when used from chrome privileged code. As you can see the if statement comparisons can be used in order to escalate access, something known as chrome execution attack. Nevertheless, the function is sufficient enough to walk any JavaScript object. You can even make it recursive if you want to go several levels down the tree. By using this function, we can compare the namespace before and after and as such detect and locate malicious code.

This is what I believe will be one of the techniques used by anti-malware software to prevent, but mostly to detect and locate, malicious code. Nevertheless, there are always methods that can be used to overcome namespace pollution problems. One of them is to use closures. Here is an exam ple:

```
(function (window, document) {


})(window, document);
```

This technique will safely execute malicious code without the need to worry about polluting the whichever namespace, as long as the evil code that is enclosed within the closure does not modify the window or the document objects. DOM manipulation is acceptable since no one is crazy enough to check for DOM changes. The document object is far more complicated and walking it is hard.

As you can see closures can be used to hide evil code. Another way for obscuring evil code is to make use of the prototype functionalities of the interpreter. The prototype object, and several other special objects, that are enclosed within every object instance can be used to hide facilities which cannot be easily spotted by JavaScript malware detection engines. Simply put, synthetic sugar, something JavaScript has plenty of, is a perfect place for storing evil code without polluting the namespace for triggering any other canary that might be there.

Take this post and put it aside until you need it.
