Attacking x86 Windows Binaries by Jump Oriented Programming

L. Erdődi*

* Faculty of John von Neumann, Óbuda University, Budapest, Hungary
erdodi.laszlo@nik.uni-obuda.hu

***Abstract*—Jump oriented programming is one of the most up-to-date form of the memory corruption attacks. During this kind of attack the attacker tries to achieve his goal by using library files linked to the binary, without the placing of any own code. To execute attacks like this, a dispatcher gadget is needed which does the control by reading from a given memory part the address of the subsequent command and manages its execution. Besides the dispatcher gadget also functional gadget is needed to implement an attack. Since the most widely used operation system is the Windows this study introduces the execution of jump oriented attacks by an example in Windows environment.**

I. INTRODUCTION

Computer security is at the forefront of interest in industrial engineering systems and gaining more ground as attacks become more complex. Jump oriented programming (JOP) [1] is the most subtle form of the memory corruption based attacks nowadays. In the case of the memory corruption the attacker changes the ordinary run of the program (breaking of the control flow) first by exploiting program vulnerabilities. After successfully breaking the control flow the attacker is about to run his own code with the operating system. There are several methods for the breaking of the control flow most of these are based on a kind of overflow. For example those method calls are appropriate for the breaking of the control flow which do not check input. The *strcpy* function for instance copies the data at the memory address received as the second parameter without any boundary check. If the data in the second parameter is larger than the place in the first parameter, the method will overwrite data which should not be overwritten. The integer overflow or the format string type vulnerabilities provide similar results. The second step of the attack is the forcing of the operating system to execute the own code of the attacker. Among these the most well-known is the stack based buffer overflow type attack where the attacking code is directly on the stack so it is enough to direct the running of the code to the stack when breaking the control flow. The heap overflow is a well-known technique as well. The operating systems have excellent protection against this type of attack. The most effective is the *W xor X* technique where the operating system sets out some parts of the memory only to write (W) and some parts only to read (X).

The stack based overflow technique is ineffective against these protections because it is possible to place the own code on the stack but the code will not run there. On Windows operating systems this protection technique is called Data Execution Prevention (*DEP*). However the attackers developed new attacking techniques. In the case of the *return to libc* attack the attacker places the necessary parameters of a well-known and linked operating system method on the stack (stack can be written) and directs the code execution to the method address. With this method the run of an arbitrary code is not possible but a function call can be executed. An even more subtle solution is the Return Oriented Attack (*ROP*) [2]. In this case the attacker looks for shortcode blocks (gadgets) in the memory which end with a *ret* instruction. Since the instruction *ret* takes from the stack the return address of the actual method call, the attacker has only to assemble the attacking code from the short gadgets and place it appropriately on the stack. There are protection techniques also against return oriented attacks [4] [5] [6] which monitor either the frequent execution of the instruction *ret*, or try to completely filter out the instruction *ret* from the kernel. The jump oriented attack assembles the attacking code from short gadgets similar to the return oriented one [3]. However an essential difference is that the gadgets end in indirect jump or indirect call instructions (e. g. *jmp eax*). If a so-called dispatcher gadget, which gets the control after every short code block and is able to execute the series of a memory addresses stored in a table step by step, is operating in the memory then the *JOP* attack will possess the same functions as the *ROP*. The enormous advantage of the *JOP* attack that it does not use *ret* instruction for the assembly of the attack (Fig. 1).

The jump oriented attacks [1] consist of three main parts: 1. The dispatcher table containing the memory addresses of the functional gadgets in the right order of the execution (here there is no code execution). The right order depends on the logic of the dispatcher gadget. 2. The dispatcher gadget which reads the current memory address from the dispatcher table and directs the code execution to the functional gadgets 3. The functional gadgets which execute some commands and then return to the dispatcher gadget with an indirect jump instruction.

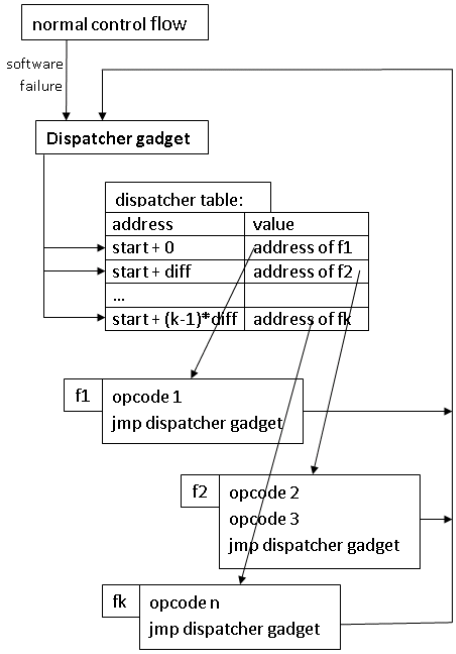

Figure 1. Jump oriented attack

Considering a symbolic harmful code *opcode1, opcode2, opcode3, ... opcode n,* the attacker looks for functional gadgets in the memory which realize the harmful attack. e.g:

:

func g1:opcode1, indirect jump to dispatcher gadget
func g2: opcode 3, indirect jump to dispatcher gadget
func gn, indirect jump to dispatcher gadget

The dispatcher gadget has to be a code block which controls the code run by changing a pointer belonging to the dispatcher table. e.g:

add memory address, difference
jump to memory address

Finally the dispatcher table contains the opcode addresses. If the dispatcher gadget gets the control with a pointer to the beginning of the dispatcher table, then every instruction of the harmful code will run in the right order.

II. THE CHARACTERISTICS OF WINDOWS BINARIES

Similar to other operating systems there is a virtual memory space belonging to each Windows process, and this contains the compiled code and the data used by the compiled code, the stack which stores the methodcall data and the linked libraries. While in the case of Linux kernel the linked libraries are stored in the *libc.so* files, the Windows operating system stores the method codes of the c library in *dll* files. Table 1 shows the analyzed memory map of a Windows XP SP2 compliled empty program.

As it is visible in this case the real code is placed to the 0x400000 address and there are other 5 *dll* linked to it. Using Windows 7 in the case of an empty file compiled by a simple compiler (lcc) the *win32 crtdll* is linked in as well, a new 6.1 version of the *user32*, the *gdi32*, the *kernel32* and the ntdll files is linked in and the *kernelbase dll* file appears with a 6.1 version too. Compilling Windows8 desktop 32 bit application the old 4.0 *crtdll* file is also among the linked in libraries, but only the *kernel32*, the *kernelbase* and the *ntdll* files are remained. A *dll* file exports and imports methods. Table 2 showes some parts of the exported methods of the *kernel32.dll.*

TABLE II.
EMPTY C PROGRAM WITH LINKED DLL-S IN WINDOWS XP

Base	Size	Entry	Name	Version
00400000	c000	00401225	helloworld	
73d20000	27000	73d21c60	CRTDLL	4.0
77d30000	90000	77d40eb9	USER32	5.1.2600
77f10000	46000	77f163ca	GDI32	5.1.2600
7c800000	f9000	7c80b436	kernel32	5.1.2600
7c900000	b2000	7c913156	ntdll	5.1.2600

TABLE I.
SOME KERNEL32 EXPORT METHODS WITH ITS ADDRESSES

Address	Type	Name
7c80a0c7	Export	WideCharToMultiByte
7c86114d	Export	WinExec
7c81cbed	Export	WriteConsoleA
7c872075	Export	WriteConsoleInputA

. Let us see the following simple but vulnerable program:

```
#include <string.h>
void func1(char * ar1)
{
        char ar2[10];
        strcpy(a2,ar1);
}
int main (int argc, char* argv[])
{
        func1(argv[1]);
}
```

The lcc compiler made machine code of the program starts from the memory address *0x401225* and goes into direction of the increasing addresses in the memory. It is also essential where the stack is stored in the memory. The stack is stored at the memory address *0x12ffcc* and it is increasing in the opposite direction of the memory. Values are placed due to the effect of the instruction push and in the case of the method call the return address of the method is stored first on the stack.

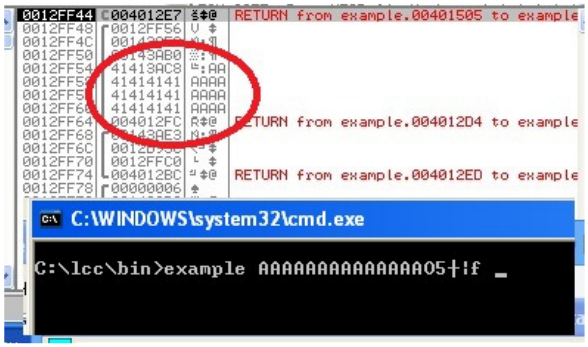

Figure 2. Overwriting return address by exploiting input validation deficiency of strcpy

The vulnerability of the program shown before can be exploited by the already mentioned stack overflow as well, if the value of the first argument is given as a value with longer arguments than 10. It is possible to direct the return of the function func1 to an arbitrary place by giving 14 arbitrary characters (56 bytes) and a 4 byte address. Due to this the control flow is broken and the composing of the well planned modification for malicious reasons can be started. Figure 2 shows the rewrite of the return address by using Ollydbg.

Assuming to have *DEP* protection of the operating system running code on the stack is not possible. It is also assumed that the operating system has protection against return oriented program (*ROP*) as well, so an exploit possibility has to be found where only the already existing code (the linked *dll* files) can be used but only in a way that too many ret instruction cannot be next to eachother because of the protection against the *ROP*. In such case the jump oriented programming is a solution. A basic condition of the executability of the jump oriented programming is that the linked code has to contain gadgets of appropriate number and variation. For functional gadgets those code parts are appropriate where the code series end with an indirect jump or an indirect call instruction. Figure 3 shows the number of the indirect jump and call in the case of the mentioned *dll* files.

It is important to note that this number contains the parts coming from the unaligned placing, so the first byte of the code can be anywhere within the *dll*.

III. THEORETICAL EXECUTION OF JUMP ORIENTED ATTACKS

For finding gadgets code parts have to be found which has an indirect jump or call at the end. Those code parts are appropriate for dispatcher gadgets where a register increases periodically and it points at a memory part which can be written by the attacker. One of the best dispatcher gadget can be found in the *crt.dll* at the address *73d3a066*:

add ebx,10
jmp dword ptr ds:[ebx]

The code parts of Table III can be appropriate for dispatcher gadget as well:

During the jump oriented attack execution of the code above the attacker can do only one thing: he can place

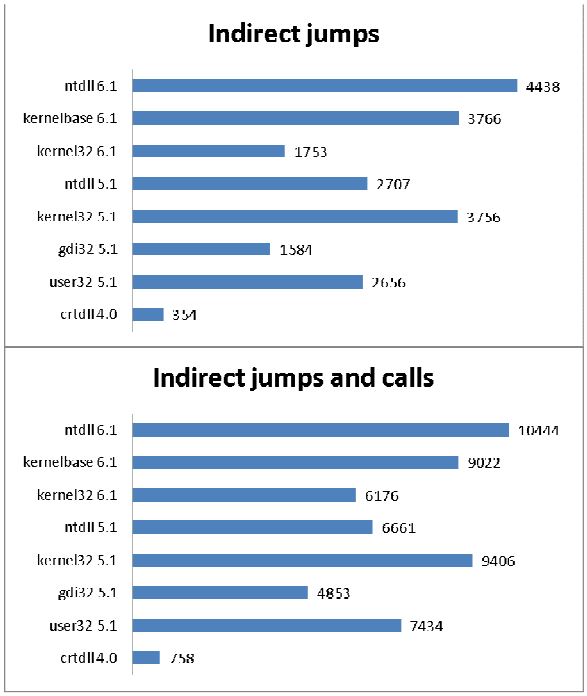

Figure 3. Number of indirect jumps and calls in Windows dll-s

TABLE III.
SOME DISPATCHER GADGETS IN WINDOWS DLL-S

File	Address	Opcode
crtdll.dll 5.1.2600	73d3a066	add ebx,0x10 jmp dword ptr ds:[ebx]
crtdll.dll 5.1.2600	73d3a0f2	add ebx,0x10 jmp dword ptr ds:[ebx]
user32.dll 5.1.2600	77d63ae9	add esi,edi jmp dword near [esi-0x75]
ntdll.dll 5.1.2600	7c939bbd	add ebx,0x10 jmp dword near [ebx]
ntdll.dll 5.1.2600	7c93c4db	sub edi,ebp call dword near [edi-0x18]
kernelbase. dll 6.2	75e6e815	sub esi,edi call dword near [esi+0x53]
ntdll.dll 6.2	77c94142	add ebx,0x10 jmp dword near [ebx]
ntdll.dll 6.2	77ca8c9	add ecx,edi jmp dword near [ecx+0x30]
ntdll.dll 6.2	77ca9dc0	add eax,edi call dword near [eax-0x18]
ntdll.dll 6.2	77cbcaca	add ebx,edi call dword near [ebx+0x5f]

data of arbitrary length and content on the stack. There is one restriction of the content: argument 1 cannot be 0 since in this case this would mean the end of a string parameter. Since the attacker can write only on the stack all the settings have to be done here to run the chosen code appropriately. These settings are the following:

- the return address of *func1* has to be overwritten so that the dispatcher gadget should get the control
- the value of the register in the dispatcher gadget has to be set so that it should work appropriate

(according to the first mentioned dispatcher gadget this is *ebx*)

- the registers pointing at the dispatcher gadgets have to be set so that the functional gadgets could operate appropriately
- the dispatcher table which consists of the addresses of the instructions to be executed placed in the right order and distance has to be placed
- the constant strings and values used by the attacking code have to be placed on the stack

In the case of the above shown program because of the above written conditions the dispatcher table and the constant data have to be placed on the stack. The setting of the registers can be done in that case when the preferred values of the registers are placed on the stack initially and the overwritten return address is the *popad* instruction instead of the address of the dispatcher gadget. In this way the dispatcher gadget gets the control only indirectly. Executing the instruction *popad* as first the initial values of the registers can be set but the control in this case does not get to the dispatcher gadget. A functional gadget has to be found for the execution which sets the registers to the right value and then it jumps onto the dispatcher gadget. The following code part in the *kernel32.dll* file is perfect for this purpose:

```
0x7c87354f:    popad
               std
               jmp eax
```

In the *dll* files above there are several code parts of similar function. However there is a problem with the placing of the initial register value on the stack: the attacker can only place code part on the stack which does not contain zero byte, since the argument 1 cannot contain that. Since the dispatcher table is on the stack and its address contains zero (the initial address of the stack: *0x0012ffcc*) the setting of the initial value of the register pointing at the dispatcher table is not possible directly. Because of all these before the code execution by the dispatcher gadget two other gadgets have to be executed as well (Fig. 4):

Since the method codes in the *dll* files gets the parameters often from the stack, the functional gadgets have to be able to do the followings: writing on the stack, reading from the stack, method call execution through the register. There can be found gadgets that seem to be appropriate for these as well (Table IV).

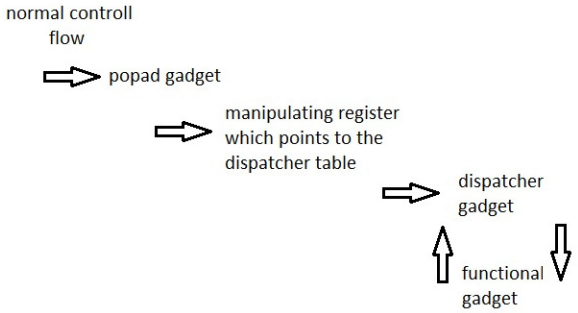

Figure 4. Gadget execution order for JOP attack

TABLE IV.
FUNCTIONAL GADGETS IN WINDOWS DLL-S

File	Address	Opcode	Function
kernel32.dll 5.1.2600	7c86114d	pop edx std jmp ecx	pop register from stack
kernel32.dll 5.1.2600	7c81ebb8	push edi jmp ecx	push register to stack
kernel32.dll 5.1.2600	7c8306f0	mov edi,ebp jmp ecx	copy register value to another
kernel32.dll 5.1.2600	7c94309ce	xchg esi,eax std jmp ecx	exchange registers
gdi32.dll 5.1.2600	77f45ce1	call esi jmp edi	call function by register

IV. ADDING USER TO WINXP BY JUMP ORIENTED PROGRAMMING

In the following the jump oriented attack is presented through an example. The possibility of applying jump oriented programming on Windows was already shown [7]. Our task is the adding of a new user to the operating system. For this the attacker has to call the *WinExec* method in the *kernel32.dll* file with the right parameters. The parameters of the *WinExec* method have to be placed on the stack. The first parameter is a zero the second is a memory address pointing at the string. The string contains the instruction which in this case is the following: *"net user PUser Passwd /ADD"*. According to these the aim is the execution of the following methods:

push 0
push address of "net user PUser Passwd /ADD"
call 0x7c86114 (address of winexec)
push 0
call 0x7c81caa2 (address of exit process))

Since both the dispatcher table and the method string can be placed only to the stack, the following gadgets have to be executed:

initial gadgets:

popad: Set registers initially, because of the zero byte in the address of *ebx*, *ebx* is set the address of the dispatcher table minus the value of *ecx*

add ebx,ecx: set *ebx* to the address of the dispatcher table

dispatcher gadget:

add ebx,0x10: increase the dispatcher table pointer

*jmp dword ptr [ebx]:*jump to the current address

functional gadgets:

pop eax: sets *eax* to the address of *Winexec*

add esi,edi: sets esi to the address of the command string "*net user /ADD*"

xor edi,edi: set zero for register *edi*

push edi: places zero on the stack

push esi: places the address of the command string on the stack

xchg esi,eax: exchange the values of *esi* and *eax*, now *esi* holds the address of *WinExec*

mov edi, ebp: sets the address of the dispatcher gadget to edi

call esi: execute *WinExec* with the appropriate arguments

xor edi,edi: set zero for register *edi*

push edi: places zero on the stack

pop eax: set *eax* to the address of *ExitProcess*

xchg esi,eax: exchange the values of *esi* and *eax*, now *esi* holds the address of *ExitProcess*

mov edi, ebp: sets the address of the dispatcher gadget to *edi*

call esi: execute *ExitProcess* with the appropriate arguments.

As it can be seen the last 4 functional gadget realize the method call. Theoretically arbitrary (the limit is the stack size) method call can be executed after eachother in this way by poping the method address from the stack. The address of the methods to be executed has to be placed in the argumentum so that *strcpy* can place in the appropriate place on the stack.

Table V shows the initial register values set by the *popad* gadget, Table VI shows the dispatcher table with the appropriate functional gadget addresses for creating new user and exiting the process.

TABLE VI.
INITIAL REGISTER VALUES

Register	Value
eax	address of add ebx,ecx
ebx	address of dispatcher table - ecx
ecx	address of the dispatcher gadget
edx	not used
ebp	address of dispatcher gadget
esi	address of command string - edi
edi	address of dispatcher gadget

Thus a new user is created by jump oriented attack. Other instructions can also be executed by the presented solution, the gadget catalogue is large enough, the *ebx* register points at the dispatcher table of which the size can be increased. There can be found gadgets executing conditional instructions and with them the jump oriented attack will be real Turing complete also on Windows system.

SUMMARY

Arbitrary codes can be executed with legitimate program by jump oriented attacks if the control flow of the legitimate program is broken by a vulnerable code. The *DEP* protection is ineffective against these types of attacks since in this case the attacker does not place own shellcode in the memory, the attacker uses the already in the memory existing code parts (gadgets). Contrary to the *return to libc* attack the *ROP* can be used for executing Turing complete programs. The surveillance of the *ret* instructions developed against return oriented programming is useless against the jump oriented programming attacking technique. The present study provides the assembly of a simple attacking code on Windows 32 system, which adds a new user to the operating system. As jump oriented attacks are more and more ubiquitous it is important to provide computational intelligence contexts [8][9].

TABLE V.
DISPATCHER TABLE FOR CREATING NEW USER

Address from the beginning of the dispatcher table	Value	Opcode	Function
0x00	77d65dda	pop eax std jmp ecx	sets eax to WinExec
0x10	77d5fa07	add esi,edi jmp ecx	sets esi to command string
0x20	77d482f6	xor edi,edi jmp ecx	zero edi
0x30	7c81ebb8	push edi jmp ecx	push zero on the stack
0x40	77d62d94	push esi std jmp ecx	push command string on the stack
0x50	7c9409ce	xchg esi,eax std jmp ecx	sets esi to WinExec
0x60	7c8306f0	mov edi,ebp jmp ecx	sets edi to dispatcher gadget
0x70	77f45ce1	call esi jmp edi	execute WinExec
0x80	77d482f6	xor edi,edi jmp ecx	zero edi
0x90	7c81ebb8	push edi jmp ecx	push zero on the stack
0xa0	77d65dda	pop eax std jmp ecx	sets eax to ExitProcess
0xb0	7c9409ce	xchg esi,eax std jmp ecx	sets esi to ExitProcess
0xc0	7c8306f0	mov edi,ebp jmp ecx	sets edi to dispatcher gadget
0xd0	77f45ce1	call esi jmp edi	execute ExitProcess

REFERENCES

[1] 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.

[2] R. Roemer, E. Buchanan, H. Shacham and S Savage, "Return-oriented programming: systems, languages and applications" *ACM Transactions on Information and System Security,* Vol. 15, No. 1, Article 2 pp:1-34, March 2012

[3] P. Chen, X. Xing, B. Mao, L Xie, X. Shen and X. Yin, "Automatic construction of jump-oriented programming shellcode (on the x86) *ASIACCS '11*, Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ACM New York, NY, USA pp. 20-29, March 2011.

[4] L. Davi, A. Sadeghi and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks" *ASIACCS '11*, Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ACM New York, NY, USA pp. 40-51, March 2011.

[5] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh and D. Ponomarev, "Branch regulation: low-overhead protection from code reuse attacks", *ISCA '12 Proceedings of the 39th Annual International Symposium on Computer Architecture*, IEEE Computer Society Washington, DC, USA, pp 94-105, 2012.

[6] J. Li, Z. Wang, X. Jiang, M. Grace and S. Bahram, "Defeating return-oriented rootkits with "return-less" kernels", *Proc. of the 5th ACM European Conference on Computer Systems* , Paris, France, pp. 195-208, April 2010.

[7] J. Min, S. Jung, D. Lee, T. Chung, "Jump oriented programming on Windows platform (on the x86)", *ICCSA 2012, Part III, LNCS 7335, pp. 376-390*, Springler Verlag, 2012

[8] J. Tick, J. Fodor, "Fuzzy implications and interface processes", *Computing and Informatics* 24: (6) pp. 591-602, 2005

[9] J. Tick, "Potential application of P-graph-based workflow in logistics", In *Aspects of computational intelligence: Theory and Applications: Revised of selected papers of the 15th IEEE International Conference on Intelligent Engineering Systems (ed: Ladislav Madarász, Jozef Zivcak) INES2011*, pp. 293-303, Topics in Intelligent Engineering and Informatics) Springler Verlag, 2012