Finding dispatcher gadgets for jump oriented programming code reuse attacks

L. Erdödi

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

Abstract—One of the most dangerous forms of the memory manipulation based attacks is the code reuse based attack type. In this type of attack the malwares do not need to place own malicious code in the memory space, they use the already linked code to achieve the aim. The present study discusses a critical part of the jump oriented programming which is nowadays the most up-to-date memory manipulation attack type. The controlling element of the jump oriented attacks is the so-called dispatcher gadget which controls the creation of the malicious code from the available legitimate code patches. The use of the dispatcher gadgets has already been introduced on 32bit Linux systems but generally the search algorithm and the classification are still open questions. This study presents the dispatcher gadgets and their characteristics found in the basic dll files of the windows 32 bit system and in the 64 bit Linux libc files.

I. INTRODUCTION

Attacks based on memory manipulation are serious threat nowadays. Memory based attack is called that when the malicious attacker exploits an error of the compiled code and through this the running of the machine code is switched to a route which is not included in its designated function. After the normal way of the code execution is broken the attacker is about to execute his own code. Thus the memory based attacks have two steps: 1. Breaking of the normal control. 2. Execution of the own code for damaging.

Several methods have already been developed for the code breaking such as the overflow based array overwrite e.g. the strcpy method call copies the content of the first argument into the second argument without any checking of the available space at the second argument. If the data size of the first argument is larger than the second argument’s size it can happen that a data is overwritten within the process and it changes the original running of the code. The integer overflow or the format string can also lead to similar results. The present study does not discusses the change of the code running instead a special problem of the previously mentioned step 2 (execution of the own code) is discussed in details.

To summarize the possible ways of forcing a program which looks to operate good to execute a harmful code the operation of the stack based processor architecture has to be considered. In the case of the stack based operation at each method call the program places the return address to be considered on the stack. There are several other data on the stack as well e.g. the local variables of the method. So if the return address is changed due to the previously mentioned array overwrite the ordinary operation of the program is changed. In the stack overflow attacks the return address of the method exit is directed back to the stack. During the process of the array overwrite not only the return address of the method is overwritten but also the own harmful code is placed on the stack. Thus with the direction of the return address back to the stack the second step is done at the same time since the own code is written into the program. The stack overflow has been a serious problem since the 90s so several defense methods have been developed against this type of attack. The most effective method is the so-called W xor X defense. The W xor X designates the memory regions either only for write (W) or only for execute (X). In this way the stack overflow attacks can easily be avoided, since if the stack can be written the harmful code can be placed in it but it cannot be executed. However the code which can be run that cannot be overwritten.

From the point of view of the attacker the most effective solution against the W xor X defense is the code reuse. If it is not possible to execute a code the already existing code to be executed has to be forced to run in a given order. In the case of the return to libc attack the return value of the method is directed to a method of a linked libc code part (e.g. execve). The so-called return oriented programming (ROP) and jump oriented programming (JOP) attacks are even more subtle than those mentioned before.

In the case of the return oriented attacks [2] the harmful code is built up from small libc code blocks (gadgets) and each gadget has a ret instruction as an end. The ret instruction takes a return address from the stack and goes on with the code execution from that point, thus from the combination of the memory addresses and method parameters placed in the stack an arbitrary code block (ROP is Turing complete) can be executed without any self written code. So in the case of the ROP the harmful code to be executed is built up from small legitimate code blocks.

Similar to the above written one but a well applicable method even in the case of defense against the ROP [4] is the jump oriented programming which is introduced bellow.

The jump oriented attacks [1] consist of three components: 1. the dispatcher table which contains the necessary memory addresses in the right order for the code 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 next memory address from the table and directs the code execution to the right place with an indirect jump or an indirect call. 3. the functional gadgets which execute a command and the return to the dispatcher gadget.
Here is a symbolic harmful code which the attacker wanted to run:

\[ \text{opcode}_1, \text{opcode}_2, \text{opcode}_3, \ldots, \text{opcode}_n \]

The functional gadgets are code blocks which contain a part of the above shown code series with an indirect jump at the end. E.g.:

\[
\begin{align*}
\text{fg 1.:} & \quad \text{opcode}_1, \text{indirect jump to dispatcher gadget} \\
\text{fg 2.:} & \quad \text{opcode}_2, \text{opcode}_3, \text{indirect jump to dispatcher gadget} \\
\vdots & \quad \vdots \\
\text{fg k.:} & \quad \text{opcode}_n, \text{indirect jump to dispatcher gadget}
\end{align*}
\]

The dispatcher gadget is a code block which controls the code run by changing a pointer belonging to the table. E.g.:

\[
\begin{align*}
\text{add memory address, difference} \\
\text{jump memory address}
\end{align*}
\]

Finally the dispatcher table contains the opcode addresses:

\[
\begin{align*}
dt \text{address} + 0: & \quad \text{address of fg 1.} \\
dt \text{address} + \text{diff}: & \quad \text{address of fg 2.} \\
\vdots & \quad \vdots \\
dt \text{address} + (k-1)\times\text{diff}: & \quad \text{address of fg k.}
\end{align*}
\]

It is visible from the above written that if the dispatcher gadget gets the control with a pointer to the beginning of the dispatcher table, then every opcode of the harmful code will run in the right order.

II. GADGET SEARCHING FOR THE JOP ATTACKS

A. Requirements of dispatcher gadgets

The requirements of the dispatcher gadgets are introduced in the following. The dispatcher gadget has to possess a pointer to the dispatcher table and the pointer has to proceed in a certain direction on the table. It can be written in general form like this:

\[
\begin{align*}
\text{pointer} = f(\text{pointer}) \\
\text{indirect jump to pointer address}
\end{align*}
\]

In the first command the pointer steps on the dispatcher table according to an arbitrary logic and in the second step it jumps to the next functional gadget. Variable value can be stored in registers or under a given memory address. If the pointer of the dispatcher table is under a given memory address, the machine code dispatcher can be rather complicated, e.g.:

\[
\begin{align*}
\text{inc} [0x34444422] \\
\text{jmp} [0x34444422]
\end{align*}
\]

The two commands above are not valid assembler codes, of course. To achieve similar functionality a command series should be found in the memory which runs into the place of a memory address of a given memory address (double indirection). Due to the complexity of the necessary code this theoretical case is not detailed here. It is much more important to find dispatcher gadgets which has a pointer that is a register:

\[
\begin{align*}
\text{reg} = f(\text{reg}) \\
\text{jmp} [\text{reg}] \text{ or call } [\text{reg}]
\end{align*}
\]

In the case shown above there are several gadgets for the indirect jump command in the libc code, so in the following the finding of these gadgets will be introduced. Analyzing the first part of the gadget the command \( \text{reg} = f(\text{reg}) \) has to move the pointer of the dispatcher table. This can be reached in several theoretical ways:

\[
\begin{align*}
\text{inc} \text{ reg} \\
\text{dec} \text{ reg} \\
\text{add} \text{ reg, value} \\
\text{sub} \text{ reg, value} \\
\text{mov} \text{ reg, reg+value} \\
\text{mul} \text{ reg, value} \\
\text{mov} \text{ reg, [reg]}
\end{align*}
\]

The last version \( \text{mov} \text{ reg, [reg]} \) realises a linked list. There is the same problem with this and the \( \text{mul} \text{ reg, value} \) commands as well: the dispatcher table is scattered in the memory. Both the return oriented and the jump oriented attacks are favorable and suitable to apply if the memory has \( \text{W xor X} \) protection. In this case it means that each element of the dispatcher table is jumping in the memory in case of the multiplication and the linked list as well, so each element should be on a place which can be written on. Another problem is that the dispatcher table has to be filled with the right memory addresses before running the malicious code. Theoretically it is possible: e.g. with format string vulnerability it is possible to write to an arbitrary memory place, but the practical realization is rather complicated since each functional gadget address has to be written to a completely different place. From practical point of view those dispatcher tables are good to use which store the functional gadget addresses in a continuous memory place and beside this to the dispatcher gadgets of this type of tables an incremental pointer which is increasing or decreasing belongs as well. When changing the pointer of the dispatcher gadget it has to be considered that the change should not be less than the length of one memory address. Of course the commands \( \text{inc} \) and \( \text{dec} \) can be excluded as well.

What is the situation with the length of the dispatcher gadget? The optimal case is if the gadget contains only two commands e.g.:

\[
\begin{align*}
\text{add} \text{ reg, value} \\
\text{jmp} [\text{reg}]
\end{align*}
\]

Of course theoretically there is no difference compared to the previous example in the following case either:

\[
\begin{align*}
\text{add} \text{ reg, value} \\
\text{nop} \\
\text{nop} \\
\text{jmp} [\text{reg}]
\end{align*}
\]
In the reality it is not the length of the gadget which counts, the important thing is not to spoil the gadget between the \( \text{reg} = f(\text{reg}) \) and the \( \text{jmp}[\text{reg}] \) and to influence as less thing as possible.

Looking at the dispatcher gadgets bellow:

```
add eax,10
ret
jmp [eax]
```

```
add eax,10
mov eax,345
jmp [eax]
```

In the two cases above the first and the last row of the dispatcher gadget is perfect but the command in the middle spoils the gadget. In the first case the execution jumps out of the gadget due to the command \( \text{ret} \), while in the second case the problem is that the value of the dispatcher pointer changes. According to these the dispatcher gadget has to fulfill two essential requirements: if the gadget contains more than two commands then the command in the mid part must not be \( \text{ret, call, jmp} \). It is unfavorable to have jump with condition (\( \text{loop, jz, je} \), etc.) because in these cases it has to be considered that the functional gadgets should return with the right variable values. The second requirement concerns to the dispatcher table pointer. The mid command (if there is or are any) must not change the content of the register that controls the jump. The same requirement concerns to functional gadgets as well.

Theoretically there can be cases where the above written are not fulfilled, but still they are still good for dispatcher gadgets.

```
add eax,10
mov ebx, eax+12
jmp [ebx]
```

Above the \( \text{eax} \) register can be regarded as dispatcher table pointer, the value of \( \text{ebx} \) can be changed arbitrary in the functional gadgets. Another interesting possibility:

```
add eax,10
jmp somewhere
...
wherever:
jmp [eax]
```

In this case there is a perfect dispatcher as well with the difference that the gadget is present in the memory in two parts.

The requirements of the dispatcher gadgets has to be investigated from the point of view of the functional gadgets as well. The dispatcher gadget must not spoil the operation of the functional gadget. So the mid commands of the dispatcher gadget must not change the already set values, unlike in the example bellow:

```
functional gadget:
mov ecx,4
jmp dispatcher table address
```

**B. Requirements of functional gadgets**

The functional gadgets have to fulfill two requirements: they have to jump back to the dispatcher gadget after the executed function, and above this they must not spoil the variables used in the dispatcher gadget (in a simple case it is only the dispatcher table pointer).

Taking into account the requirements above there can be many code blocks in the libc files which are good for dispatcher and functional gadgets. The question is how to find these blocks automatically and classify them and how to decide with an algorithm which suits the best for a certain task.

![Finding dispatcher gadgets in user32.dll going backward direction of indirect jumps or calls](image)

**C. Classification of dispatcher gadgets**

Obviously the situation is different in the case of aligned and unaligned machine codes so the two cases has to be handled separately. In the unaligned case the beginning address of the gadgets can be arbitrary. In the aligned case only those memory addresses can be analyzed which are beginning addresses of a real libc assembler command. In both cases one gadget can be written with two variables: the beginning address and the number of the commands.

In the case of dispatcher gadgets the higher the number of the commands the better the chance of that the signed dispatcher gadget does not fulfill the requirements above. However considering the performance of the available hardware it is not a serious problem to check each case until a certain command number it is still better to start the search from the end. The sequence of the commands can be illustrated in graphs [8]. In the present study the search is done according to the following: indirect jump and indirect call commands were searched for in the libc code blocks and then from this point the possible gadgets were...
made by counting back the commands (Fig 1). Those which did not meet the requirements were excluded.

Since the requirements of the dispatcher gadgets can be fulfilled by more than one libc code blocks, a tool is developed to rank them. It is obvious that the length of the gadget is not an important factor since the mid commands can be neutral to everything. The most important factor is the number of the related registers. The continuously modified registers in the dispatcher gadget cannot be used in the functional gadgets so the higher the number of these, the less usable the dispatcher gadget is. The second aspect is the used register itself. Of course the use of a less used register as a dispatcher table pointer e.g. esi or edi is much more favorable than the use of eax or ecx. According to these the following order has been set between the registers:

r8-r15, new registers
esi, edi, ebp(esi, rdi, rbp)
eax, ebx, ecx, edx, esp (rax, rbx, rcx, rdx, rsp)

According to these the most favorable case is when the dispatcher gadget influences one register value and this is e.g. r9x. Finding of the best dispatcher gadget can be a part of the implementation of an automatic jump oriented attack.

III. THE RESULTS OF THE GADGET FINDING IN THE CASE OF DIFFERENT PLATFORMS

The automatic dispatcher finding algorithm was tested on 32 bit windows binary files (Windows XP SP2) and on a 64 bit Linux libc file (Ubuntu 12.04). Table 1. shows the most suitable dispatcher gadgets.

In the case of Windows XP system a really good dispatcher has been found which uses only the register esi. In the case of Linux 64 bit libc.so file the most suitable gadget influences two registers.

Since the functional gadgets can modify the dispatcher gadget by evaluating conditions, thus they can write an arbitrary algorithm [1] [7].

Since there are suitable dispatcher gadget code blocks in the libc files of the investigated operating systems, the strong defense against ROP [6] can be useless in the case of JOP attacks. The defense against the jump oriented programming should consider its introduced specialties as well [5].

IV. CONCLUSION

Information security is one of the key questions of information technology. [9][10] This study discusses the finding of dispatcher gadgets in shared libraries which are necessary for the code reuse jump oriented attacks. The dispatcher gadgets can be present in existing libc codes. For the finding of these gadgets it was searching for code blocks which meet certain requirements. Then these code blocks (which met the necessary requirements) were classified from the point of view of their usability. It has been concluded that the analyzed dll and so files contain code blocks which are well usable for jump oriented attacks. It is also concluded that from the finding process of these code blocks an algorithm is possible to be developed. With the classification of the dispatchers a method is set which is able to select the most suitable dispatcher gadget. All these lead to a conclusion that with

<table>
<thead>
<tr>
<th>File name</th>
<th>Dispatcher gadget</th>
<th>Assembly</th>
<th>Pointer to dispatcher gadget</th>
<th>Registers to influence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libc-2.15.so</td>
<td>add edi, edx jmp dword far [rdi]</td>
<td>rdi</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>user32.dll 5.1.2600</td>
<td>add esi,0xfe call dword far [esi+0x33]</td>
<td>esi+0x33</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>crtdll.dll 5.1.2600</td>
<td>add ebx,0x10 jmp dword ptr ds:[ebx]</td>
<td>ebx</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>user32.dll 5.1.2600</td>
<td>add esi,edi jmp dword near [esi-0x75]</td>
<td>esi-0x75</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ntdll.dll 5.1.2600</td>
<td>add ebx,0x10 jmp dword near [ebx]</td>
<td>ebx</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ntdll.dll 5.1.2600</td>
<td>sub edi,ebp call dword near [edi-0x18]</td>
<td>edi-0x18</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>kernelbase.dll 6.2.9200</td>
<td>sub esi,edi call dword near [esi+0x53]</td>
<td>esi+0x53</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ntdll.dll 6.2.9200</td>
<td>add ebx,0x10 jmp dword near [ebx]</td>
<td>ebx</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ntdll.dll 6.2.9200</td>
<td>add ecx,edi jmp dword near [ecx+0x30]</td>
<td>ecx+0x30</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ntdll.dll 6.2.9200</td>
<td>add eax,edi call dword near [eax-0x18]</td>
<td>eax-0x18</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ntdll.dll 6.2.9200</td>
<td>add ebx,edi call dword near [ebx+0x5f]</td>
<td>ebx+0x5f</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
the knowledge of the malicious code to be executed and the libc files the development of a jump oriented attack code can be automated.

REFERENCES


